UiO : Department of Informatics
University of Oslo

Rapid Prototyping CNC

Just press "print"

Thomas Knaplund Benjaminsen

Master's Thesis Spring 2015

Rapid Prototyping CNC

Thomas Knaplund Benjaminsen

15th May 2015

Abstract

Rapid prototyping helps people turn ideas into physical objects in a fast and uncomplicated manner. Today, the most popular way of conducting rapid prototyping is by far 3D printing.

This thesis questions the possibility of creating a system that can make CNC milling machines utilize rapid prototyping with the push of a button. To explore this, a system called BabyMill, consisting of two parts has been made. The practical part has examined different work holding solutions that requires the least possible interactions from the user. The software part of the system has looked at ways to read STL files and inspected different milling strategies and implemented these to calculate tool paths with as little user inputs as possible.

The BabyMill system has successfully milled parts in both foam and aluminium, with very promising results. This shows that it can create parts with both the ease of use, accuracy and finish that of a FDM 3D printer. Although maybe not as convenient to use yet, the BabyMill system shows great potential when it comes to making CNC milling machines as user friendly as 3D printers.

Contents

List of Figures

List of Tables

Glossary

RP Rapid Prototyping

CNC Computer Numerical Control

CAD Computer-Aided Design

STL STereoLithography

CAM Computer-Aided Manufacturing

DIY Do It Yourself

FDM Fused Deposition Modeling

NC-code Numeric Control code

G-code A language for describing CNC tool paths

UGS Universal G-Code Sender

IDE Integrated Development Environment

PBP Powder Bed Printing

SLS Selective Laser Sintering

Acknowledgement

I would like to thank my supervisor, Mats Høvin, for the support and advice provided throughout the work of this thesis.

I would also like to thank my fellow students at the ROBIN group for interesting discussions and ideas, and for making each day at school brighter. In the end I would like to express my gratitude to the espresso machine, which has given me energy throughout this thesis.

Part I

Introduction

Chapter 1

Introduction

1.1 Introduction

Through 3D modelling people can create prototypes of any shape and size, unleashing their creativity and imagination upon the world. One thing is looking at these on a computer screen, another entirely, is having a physical object to admire, caress and maybe even make good use of. Rapid prototyping is the fastest way of achieving this. It is just as the name implies; a quick and easy way to create a prototype. 3D printing and prototyping is something that has really taken off in the last couple of years. It is no longer reserved for specialists and engineers, as 3D printers can be bought by the general public in retail stores or on-line. 3D printing utilizes rapid prototyping and this is one of the reasons it is so popular. It makes users able to push a button, walk away and come back to evaluate the result. Computer controlled machining known as CNC is on the other hand something most people still are oblivious of, even though the process itself is just as simple as 3D printing. Instead of building the prototype from the ground up by binding material together, the prototype is produced by removing material from a solid piece.

However, CNC machining in general is in contrast to 3D printing *not* rapid prototyping. This is because it requires some extent of user inputs and interactions both before and during the process. This thesis will examine the possibility to create a system that enables simple CNC milling machines to be as easy and convenient to use as 3D printers.

1.2 Motivation

A common scenario where a system such as this can come in handy is if someone, either a student at school or a hobbyist in a small workshop, created a small part in 3D modelling software that has to be realized in physical form. A 3D printer is available, but preferably the part needs to be in a more robust material, like metal, and those kinds of printers are way

out of reach. There is also a CNC milling machine available, but it is both time consuming and a lot of hassle for one small part. The student might also lack the required knowledge to set up and operate the CNC milling machine, while the employees that possess that knowledge do not have time or are otherwise engaged. A solution where all they need to do is upload a file containing the part, insert some material and press a button could really save them both time and frustration.

1.3 Goals of the thesis

The objective of this thesis is to see if a system can be developed that makes it possible for simple 3-axis CNC milling machines to utilize some degree of rapid prototyping. At the same time it should be as uncomplicated as possible such that anyone can use it. Perhaps by keeping the user inputs to the absolute minimum, both these criteria can be met. Maybe the only options the user gets is to upload a computer file and press a button. The system should still deliver a satisfactory level of quality to the prototypes produced, and should be able to mill metals like aluminium.

The system will consist of two parts:

- **Physical set-up**

  Explore different solutions to keep physical interaction with the milling machine to a minimum, by finding an easy way for users to insert material and retrieve the finished part.

- **Software**

  Create software that analyses the computer file provided by the user. Examine different milling strategies and implement these into the software. It should with as few user inputs as possible be able to create the necessary code and transferring it to the milling machine that mills out the part hiding in the computer file.

1.4 Outline of the thesis

This thesis consists of 7 chapters divided into three main parts: introduction, the project and conclusion.

- The introduction part consist of two chapters. First there is an introductory chapter explaining the objective and motivation for this thesis. This is followed by a background chapter that provides some relevant information on the topics of this thesis and some of the previous research and work on the subject. The tools and programs used to realize the thesis is also listed here.

- The project part includes the planning of the project, implementation and testing. This will include the process of designing the set-up and choosing strategies and milling techniques for the software. The implementation of these, and testing of the whole system.
- Last part is the conclusion where results from different tests are presented and analysed. This is followed by a chapter that discusses the results, along with a conclusion and suggestions for future work.

Chapter 2

Background

2.1 Rapid prototyping

Rapid prototyping all begins with a description of a model saved on a computer file. A 3D CAD(Computer Aided Design) geometry description to be precise. The concept of RP is simple, after CAD, press "print"[18].

2.1.1 The STL file format

The main geometry file format used for any rapid prototyping process is the STL(STereoLithography) file[35]. An STL model consists of a triangular polygon mesh. Since it describes the real 3D CAD model with triangles, an easy shape, like a cube, can be achieved by putting 12 triangles together in a mesh as figure 2.1 shows. A more complex shape however, e.g. a sphere, will need an infinite amount of triangles to be rendered correctly. Since this of course is impossible, it becomes a close approximation of a sphere.

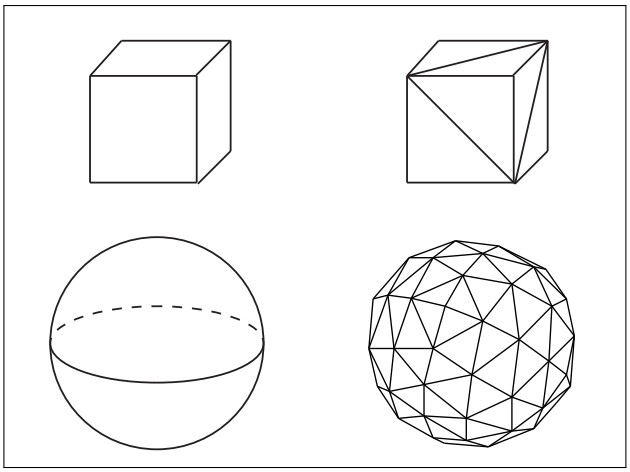

Figure 2.1: The STL approximation of a cube and a sphere.

2.1.2 Tool path generation

The next part of the rapid prototyping process is to generate tool paths from the STL file. This is done by slicing the triangular polygon mesh into horizontal layers [23]. The more layers, the better the geometry approximation, but building the prototype will take longer time. To find the outline of the model in the different layers, each polygon is converted to a contour line at the intersection between the polygon and the slicing plane, as figure 2.2 shows. All contour lines are then sorted and the interior/exterior area of each contour is found[23]. The tool paths filling out the interior(for 3D printing) or exterior(for CNC machining) for each layer are then generated, and written to a file in NC-Code(Numeric Control Code). The common name for the most widely used numerical control programming language is G-code.

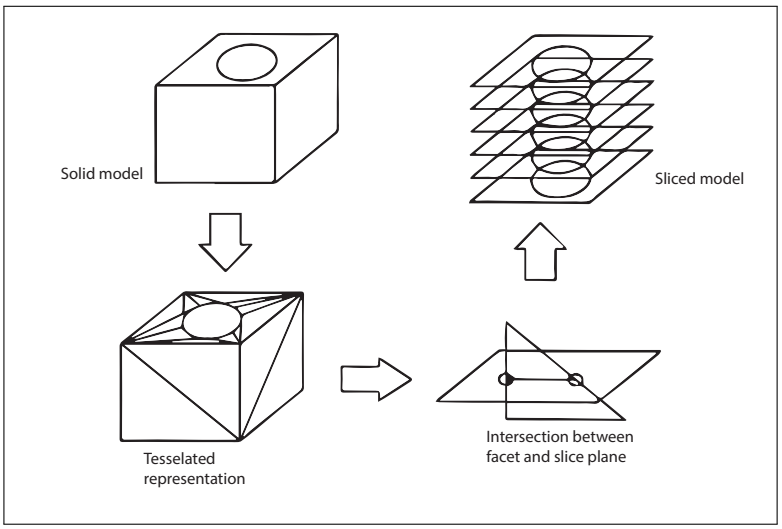

Figure 2.2: Slicing the STL model[46].

2.1.3 G-code

G-code was originally defined by the Electronics Industry Association (EIA) in the 1960's[19]. G-code is basically instructions to tell the machines what to do, i.e. how to move, how fast and through what path. In its simplest form G-code can be used to specify linear interpolation. Run each X, Y, Z motor to set the tool tip to the specified position (mm or inch), with a speed such that all motors will finish at the same time. The speed of the tool tip along the line is specified by an F command, and can be interpreted as mm/min[19]. E.g. the commando G01 X30.0 Y40.0 Z50.0 F200 will move the tool tip linearly from the current x, y, z position to the new position x = 30.0 mm, y = 40.0 mm, z = 50.0 mm with a line

speed of 200 mm/min. To minimize file size, G-code is modal, values that are not specified on the command line are assumed to be kept at the old values[19].

2.2 3D printing

A prime example of rapid prototyping is 3D printing. There are numerous different kinds of 3D printers, ranging from multi million commercial ones, to simple hobby versions[42]. Common for the majority of printers is that the prototype is manufactured one layer at a time. It can be compared to regular 2D printing used to print on paper, only with one extra dimension, hence the name 3D printing. As mentioned there are many different printers, resulting in many different printing techniques. Many of them are so-called additive manufacturing processes. Some of these are: FDM(Fused Deposition Modelling), PBP(Powder Bed Printing) and SLS(Selective Laser Sintering).

2.2.1 Fused Deposition Modelling

FDM can easily be understood as drawing with a very precise hot glue gun[25]. The process begins with G-code generating software that determines how the extruder, the part depositing material, will draw out each layer to build up the model as figure 2.3 shows.

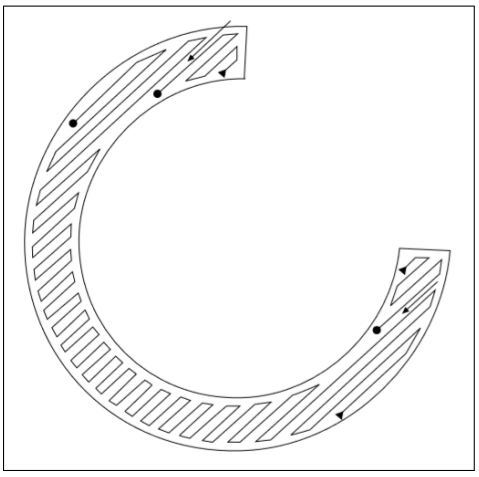

Figure 2.3: Pattern employed by FD processing to build a layer of a "C" ring[1].

The actual printing process works by using a motor to feed a filament with material through a heating element that melts it at a temperature that typically ranges between 170 and 300 degrees Celsius, depending on the type of material being used[25]. The filament emerges molten and

quickly hardens to bond with the layer below it. The nozzle and/or the build platform moves in the X-Y (horizontal) plane before moving in the Z-axis (vertically) once each layer is complete. In this way, the model is built one layer at a time from the bottom upwards[25]. Some prototypes have high complexity and require some extent of support in order to avoid extruding material into thin air. Because of this, FDM printers may utilize two nozzles. One nozzle produces the material used for the model itself, while the other produces support material. There are printers with more than two nozzles to get alternative colours. If the object was printed using support material, after the printing process is complete, the support is snapped off or dissolved in solvent leaving behind the finished model[25].

FDM uses high quality industrial grade plastics such as ABS, polycarbonate(PC) etc. to produce strong, robust parts suitable for functional parts. While there are many advantages, there are some disadvantages[10]. If support material has been used, it can be a hassle to get rid of it all. Also, FDM does not have the best surface finish. Since it lays down layers like a glue gun, it is possible to see the lines of each layer quite easily. If surface finish is important, PBP and SLS can be an alternative.

2.2.2 PBP and SLS

Powder bed printing (PBP) and selective laser sintering (SLS) are two very similar printing techniques. Like FDM, the prototype to be printed is built up from thin layers of a 3D model. The printer itself consists of two adjacent tanks of powder, where one of them is the build tank, and the other the powder reservoir[20]. In PBP, an inkjet print head moves across the top of the powder in the build tank, selectively depositing a liquid binding material. When the material is bound, the build tank is lowered and a fresh layer of powder is spread across the top, and the process is repeated. SLS uses laser instead of binding material, and bind the powder by local melting. When the model is complete, the unbound powder (support) is easily removed[20].

Parts made with PBP have high local accuracy and surface finish, but models may warp when post-processed[20].The parts are therefore not suitable for use as functional parts. SLS however, can make durable functioning parts with high surface finish depending on the powder used. But even though these parts have a high surface finish, it is nothing compared to what can be achieved with CNC machining[44].

2.3 CNC milling

While a 3D printer creates a prototype by adding or binding material, a CNC (Computer Numerical Controlled) milling machine does just the opposite. Starting with a solid block, it removes material to create the pro-

totype. As with 3D printers, there is a whole variety of milling machines. There are the advanced high-end ones that can make complicated and extremely detailed models out of blocks of titanium. There are also the regular basic custom built milling machines, and of course many in between.

CNC milling machines have been developed based on conventional milling machine, where the tool is moved by operating a hand wheel for each axis. The basis of adding numerical control is simple: replace the hand wheels with motors and some electronics to control the position of the tool[34].

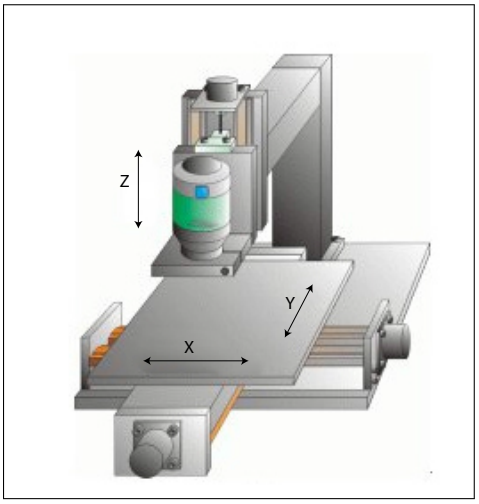

Figure 2.4: A typical milling machine[2].

Figure 2.4 shows a typical 3-axis milling machine, it can be very similar to an FDM printer in how it is put together. It has a platform where the solid block of material sits. This can typically be moved in the x and y direction. Above the block sits the milling tool that moves up and down in the Z direction to complete the 3-axis system. Removing material requires quite a bit of force, making the work piece stay put on the platform while milling can be challenging.

2.3.1 Work holding

A commonly used work holder is the vice. For CNC milling the precision requirement of the vice is normally very high, as it is in many cases used as a reference for relocating the work piece when milling on multiple faces. Depending on the precision of the CNC machine the geometrical tolerances can be as low as a few micro meters[22]. For milling soft materials and removing only small volumes of it, various types of clamps can be used. They press the work piece downward against the table so it does not move while milling[22].

2.3.2 Milling tools

Milling tools come in a range of sizes, materials, and geometry types. It is often more efficient to use a combination of different tools to achieve a detailed result[3]. Two of the main types of tools include the flat mill and ball nose. The flat mill is often used for the initial milling operations where large quantities of material is removed. In many milling operations, the cutting tool must step over and make several adjacent cuts to complete machining a feature. As a result, a small peak of material, called a scallop, will remain between these cuts[37]. Flat end mills will cut flat areas with no scallops. However, they leave a stairs-like scallop on non-flat surfaces[3]. The ball nose is typically used in the last milling operations when putting on the finishing touch. Ball nose ends will leave smaller scallops on sloped surfaces compared to flat mills, but they will also leave scallops on flat areas[3].

Each tool has its strengths and weaknesses. To get perfect results, different milling strategies have to be utilized with the different tools.

2.3.3 Milling strategies

When CNC milling it is important to optimize one or more of the following parameters[21]:

- Highest possible volume removal rate in the smallest amount of time
- Surface finish
- Tool life
- Heat
- Minimal interaction with the machine
- Safety considerations

All of the above factors are directly affected by spindle speed, feed rate and milling strategy. Spindle speed is the speed the end mill is rotating with in RPM(Revolutions Per Minute) while cutting speed is the speed of which the end mill is travelling over the surface when milling. Feed rate is the rate at which the tool will advance into the material. The feed rate that can be used, is determined by the spindle speed, the number of cutting edges(flutes) on the tool, and by the chip load[12]. The chip load is the average thickness of the chips that are cut off the work piece.

Normally a milling job is divided into two phases, roughing and finishing. Roughing is all about maximizing volume removal at the expense of surface quality. High feed rates and step-over is used. The finishing process on the other hand, uses a low step-over, typically only

0.1-0.3 mm material is removed for each pass[21]. Finishing operations may therefore be very time consuming.

When removing material there are normally many different milling motion strategies to choose from, and knowledge about things like spindle speeds, feed rates and step-over is required. A good CAM(Computer Aided Manufacturing) program will normally offer optimized motion strategies for different milling operations.

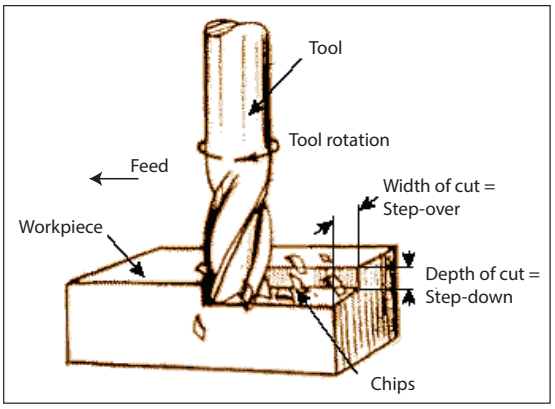

Figure 2.5: Illustration showing different milling terminology[21].

2.3.4 CAM

Before CAM, when an operator was to mill something, he entered the complete NC-code on a computer, using a plain text editor like Notepad, or a special purpose NC editor. Each movement had to be separately entered and this was of course a tremendous amount of work. Luckily, CAM software came along, software that 'automatically' generate NC program files[34].

CAM can in many ways be compared to the rapid prototyping tool path generation for 3D printing. Both start out with CAD files and generate tool paths based on these. Because CNC milling is a more complex process, most CAM programs need user inputs about feeds and speeds, cut depth and tool selection. This makes traditional CNC milling a non-rapid prototyping procedure.

2.4 Previous work

Making CNC milling machines more user friendly is not a new idea. A lot of work has already been done on the subject, and in this section we take a look at some of it.

2.4.1 Implementing rapid prototyping using CNC machining (CNC-RP) through a CAD/CAM interface

This particular approach has been realised by Dr. Matthew C. Frank at the Department of Industrial and Manufacturing Systems Engineering at Iowa State University of Science and Technology[14].

Introduction

CNC-RP or Computer numeric Controlled Rapid Prototyping through CAD/CAM is a method which enables automatic generation of process-plans for a component that is to be machined(milled). By using advanced geometric algorithms, true automatic NC code generation is achieved directly from CAD models with no human interaction, a capability necessary for a practical rapid prototyping system[14].

Most RP systems are based on the additive layer stacking process, and attempts to automate CNC machining have also been approached from the perspective of traditional machining methods, but maybe it is necessary to re-think how parts can be held, oriented, and cut[14].

The idea is to place the stock material between two opposing chucks. The material can then be rotated by a rotary indexer. For each orientation, all visible surfaces are milled by an end mill which can move in the x, y and z direction. A set of sacrificial supports will keep the part connected to the uncut ends of the stock material while milling. When all operations are complete, the supports are milled or sawed off, leaving the part free to be removed[14]. Figure 2.6 shows the process.

To achieve this, quite advanced algorithms are needed to handle steps including:

- analyse the part for an axis of rotation
- establish a work coordinate system
- attach sacrificial supports
- determine set-up orientations about the axis
- generate roughing process to remove material
- generate finishing process to create surface geometries

Finding the axis of rotation

First off, since the process requires an axis of rotation, one needs to analyse the part to determine if one exists, and if there are multiple, choose the best one. This search is based on the surface visibility; rotate around an axis

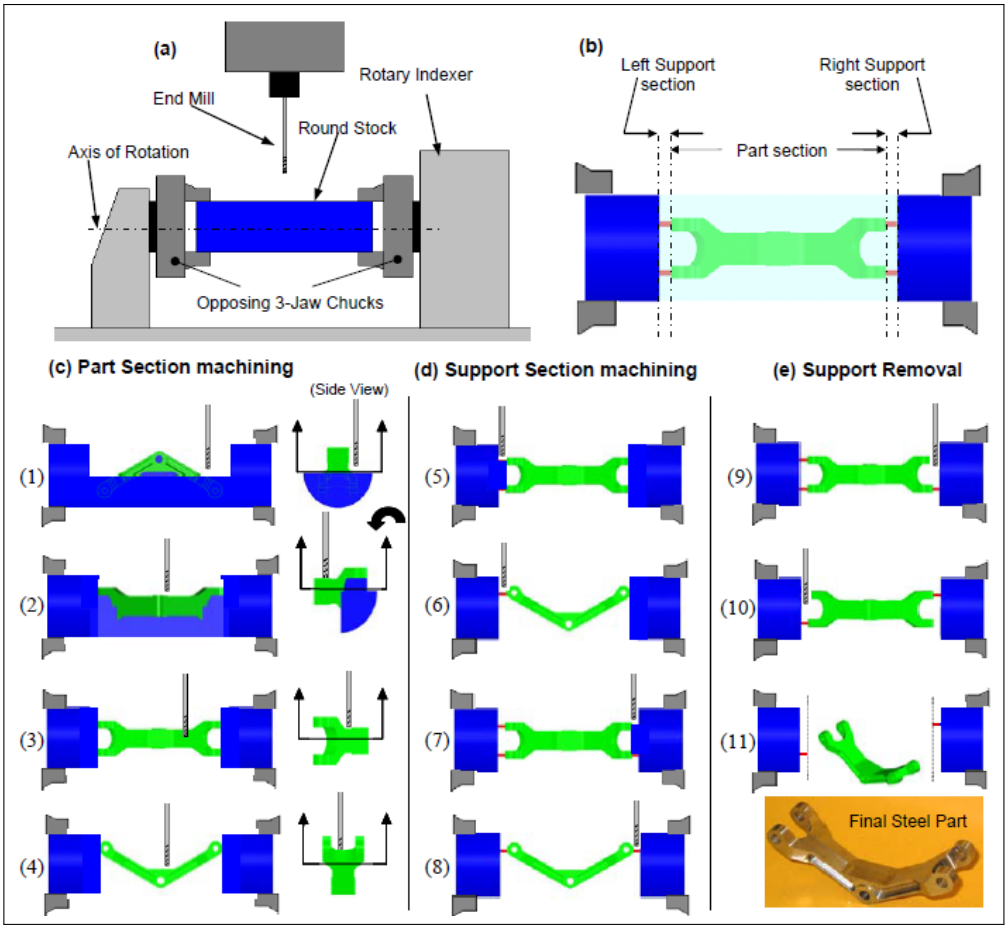

Figure 2.6: Rapid machining; (a) set up, (b) sections machining approach, (c) Part Section machining steps, (d) Support Section machining steps, and (e) Support removal steps[14]

that makes the end mill able to reach all surfaces of the part[29]. Figure 2.7 shows the visibility for a cross section of a part.

Work coordinate system

To avoid any tool collisions during processing, the coordinate system must be oriented with care. A simple translation of the part is made to shift the part to the negative x-space and centre it on the axis of rotation[14]. This ensures that the part is placed in the centre of the stock material. In addition this can ensure collision free processing by establishing a safe working space between the chucks of the fixture system.

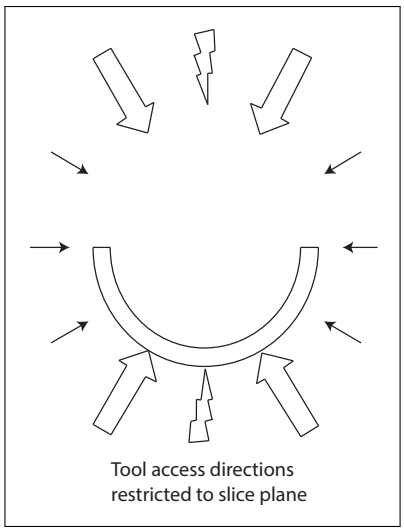

Figure 2.7: Sample model with cross section for visibility mapping[29]

Sacrificial supports

Conventional fixturting methods for CNC milling include clamps, vices etc. These approaches make it difficult to reorient the part, and multiple set-ups are required. The approach to part fitting for this research borrows from the idea of sacrificial supports. The sacrificial supports are added to the ends of the CAD model automatically, this ensures that the part is attached to the stock material throughout the milling operations, so no human interaction is needed in the milling process (unclamp, remove, replace, reclamp)[14]. A minimum of two to as many as four supports are added. Two supports are "permanent" and if possible, two more "temporary" supports are added. The concept is to limit the maximum deflection to at most, half of the required part tolerance. The sacrificial support has to be placed in such a manner that visibility to the surfaces to be milled is not occluded[14].

Set-up orientations

After supports are added, the program has to determine the orientations for machining of the part. It also has to calculate depths of cut, step-down, feeds and speeds for each orientation. In this step, a greedy algorithm is used to quickly find a set of angles of orientation required to machine the part. The orientation angle with the most visibility of the surface of the part is used as the seed value for the next operation, which is the roughing process[14].

Roughing process

The roughing tool paths are generated using a heuristic algorithm developed from numerous part trials. To prevent thin material deflects to wrap around the tool and cause failure, the roughing process need to be sequenced to a specific order of set-up orientations, so that the stock material can be removed in a controlled, incremental manner[14]. Since this is a rapid prototyping environment, the tool settings are very limited, typically one or very few are used. Tools are chosen based on their necessary criteria; long enough to reach the part surfaces. A tool with a small enough diameter must be chosen because the surface geometry of the part may contain small features. The feeds and speeds are usually the maximum that are allowable for the different materials, given the step-down amount. For each step-down the tool removes material layer by layer, in most CNC-RP tool paths, this layer depth is 0.5 mm to 1.27 mm for the roughing process, and 0.025 mm to 0.076 mm for the finishing process[14].

Surface geometries

For the next part, a visibility algorithm sorts surfaces to be milled by allocating each surface to the set-up angle that can reach the surface with minimum distance. This is calculated using depth calculations from the slice geometry of the part file. Tool paths generated for both the roughing and finishing process are generated from the original CAD model of the part, therefore the tool paths in CNC-RP are not based on an STL file, rather, from the native surface geometry[14]. This way, CNC-RP avoids approximation errors that exist in additive processes (3D printing) which calculate tool paths from STL models. The steps so far results in a concentrated set of NC-code and a set-up sheet. The set-up sheet lists the tools required, tool changer locations, and diameter and length of the stock material. To run the program, the user loads the material and NC code and press the start button, which initiates a cycle-start[14]. However, as can be seen in the flowchart in Figure 2.8, some user inputs are required to obtain the NC-code and a set-up sheet.

Conclusion

Traditional additive RP processes are infinitely capable at creating complex shapes compared to CNC machining. However, CNC-RP is usually more capable in surface finish, since in contrast to additive methods, layer depths can be set to very small values. The greatest advantage however, is that CNC-RP can use a vast variety of materials to create parts that are truly functional[14].

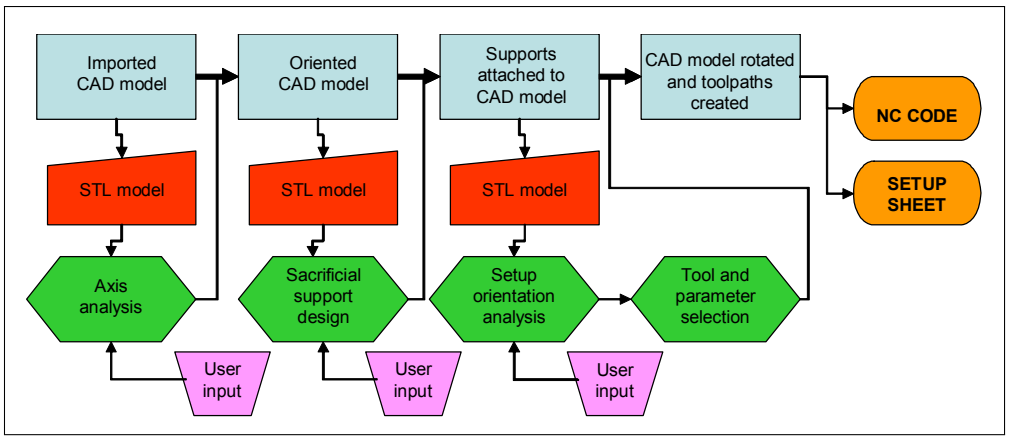

Figure 2.8: System Flowchart illustrating interaction between CAD models in the CAM system and algorithms, through the STL file format[29]

2.4.2 DeskProto

In 1996, DeskProto, developed by Delft Spline Systems located in the Netherlands, made rapid prototyping available to a much wider audience by applying 'traditional' CNC technology to this area.[35]

How does it work

Where most rapid prototyping systems are based on incremental build techniques, the DeskProto approach is decremental.[35]. The starting point for DeskProto is a STL file. Also DXF files containing "3D Faces" and VRML files can be processed[11]. DeskProto reads the 3D geometry and displays its contents. At this point it is possible to scale the geometry, translate, rotate etc. After entering some milling parameters (type of cutting tool, required accuracy, etc.) DeskProto will automatically calculate the milling paths. DeskProto comes in three versions:

- Entry Edition
- Expert Edition
- Multi-Axis Edition

The entry edition is most relevant since it is the version that requires fewest user inputs while the more advanced ones require some extent of CAM knowledge because more options are available. However, to actually mill something, additional software is required, as DeskProto only produces the G-code to mill the prototypes. DeskProto have been constantly updated since 1996, and on October 10. 2013 Delft Spline Systems releases Version 6.1 of the DeskProto 3D CAM software[9].

2.4.3 The Nomad CNC mill

Using kickstarter, Carbide 3D, a small company from the United States launched a project called The Nomad CNC Mill, and has the slogan 'The Nomad 883 is a ready-to-run CNC Mill that's at home in any environment.'[36]. Kickstarter is a global crowdfunding platform, so people all over the world can 'back' a project should they find it interesting. On May 30. 2014 the nomad CNC mill was successfully funded. The idea is to make a CNC machine that is easy to use, even for people that are not CNC machinists. The Nomad 883 is fully enclosed to control the mess and limit the noise[36]. It ships with MeshCAM to turn 3D parts into something the CNC mill can work with, and Carbide Motion to control the Nomad 883[36].

MeshCAM

MeshCAM is a 3D CAM program that translates 3D files into something that The Nomad can use. MeshCAM has been under continuous development for 10 years. The MeshCAM process consists of three steps[4]:

- Load a file from almost any CAD program
- Build an efficient toolpath with minimal input
- Save G-code that works on the CNC machine

MeshCAM works with almost every 3D CAD program by opening the two most common 3D file formats, STL and DXF[4]. A CAD program is not required, MeshCAM can open any image file (JPG, BMP, or PNG) and MeshCAM converts it to a 3D surface that can be machined directly[4]. MeshCAM has an Automatic Tool path Wizard to help create tool paths. It picks values like feed rates, speeds, depths of cut, etc. to reduce the learning curve associated with CNC machining. The user pick the cutters and tells MeshCAM the desired quality level. MeshCAM will then analyse the model to pick values to get you started. The values can be tweaked to be made better or used as-is[4]. Another feature MeshCAM has, is that it can add sacrificial supports to the part, to keep it connected to the stock material while milling if the part is too complicated for use of vices or clamps. MeshCAM has a built-in post processor to transform the G-code to work with various CNC machines. It supports lots of machine types as-is and it can be extended to support most others[4].

Carbide Motion

Carbide Motion is the software that takes the tool path (G-code) from MeshCAM and drives the nomad CNC mill. It is designed from the ground up to make things as simple as possible without limiting control. It comes

with a tool path simulator so the user can see what the part will look like before starting the job[36].

The promise Carbide 3D makes with this product is that if you can use a 3D printer, you can use The Nomad 883. The product has, as of September 11. not yet started shipping to backers of the project[47]. Something that the Nomad 833 is missing that could make it even easier to use, is a simple solution to add and remove stock material.

2.5 Tools and programs used

This section takes a brief look at the tools and programs used while working on the thesis.

2.5.1 Java

This thesis is programmed in Java. Java is a programming language and computing platform released in 1995 by Sun Microsystems[39]. Java was created with the following design goals in mind:

- **Simple, object oriented, and familiar:** Java is a simple language where the concepts of Java technology are quickly grasped. It is also object oriented from the ground up and provides a clean and efficient object-based development platform. Programmers can easily access existing libraries of tested objects that adds a range of extra functionality like I/O and network interfaces to graphical user interface(GUI) tool kits[38].
- **Sturdy and secure:** Java language is designed for creating reliable software. It provides considerable compile-time checking, followed by a second level of run-time checking. Memory management is extremely simple; objects are created with a *new* operator, and there is an automatic garbage disposal.[38]
- **High performance:** By adopting a design by which the interpreter can run at full speed without needing to check the run-time environment the Java platform achieves high performance. The automatic garbage collector runs in the background as a low-priority thread, ensuring a high probability that memory is available when required.[38]

Eclipse

Eclipse is an integrated development environment(IDE) used to code the software for this thesis. It was originally developed by IBM, but was

released as open source in 2001. Since 2004 it has been managed by the Eclipse Foundation[13] and new versions have been released every year. It contains a base workspace that allows users to gather source code files and resources and tie them together, making working on bigger projects easier.

JTS Topology Suite

The JTS Topology Suite is a Java API made by Vivid Solutions, Inc. It implements a core set of spatial data operations using an accurate precision model and robust geometric algorithms[43]. With classes for different geometries including polygons, points and lines, this API provides functions for easy computing of intersections, unions and buffers(offsets) of geometries.

2.5.2 Universal G-Code Sender

Universal G-code Sender is an open source Java based Grbl compatible cross platform G-Code sender[49]. In this thesis it is used to run G-Code generated by the main software on Grbl controlled CNC machines.

Grbl

Grbl is a free open source, high performance software for controlling the motion of CNC milling machines.[48]. It was written and released in 2009 by the Norwegian Simen Svale Skogrud. Since 2011 Grbl is pushing ahead as an open source phenomenon, driven by the community under leadership of Sungeun K. Jeon Ph.D.[48] It interprets standard g-code and has been tested with the output of several CAM toolss. Arcs, circles and helical motion are supported, as well as, all other primary g-code commands.[48]

2.5.3 CutViewer

CutViewer is a program that simulates g-code developed by Lamson Global, a small software company that focuses only on CNC simulation[15]. In this thesis CutViewer is used to simulate the generated g-code before running it on the real CNC machines. Not only does it present the final result, it also shows the movement of the tool for every line of g-code, making it easy to detect collisions and misbehaviour.

2.5.4 SolidWorks

Perhaps the most popular 3D modelling software available today is SolidWorks. Building a model in SolidWorks begins with a 2D sketch that

later is extruded into a 3D model. Used in this thesis to produce work-holding and models of varying complexity to test the BabyMill software.

2.5.5 HSMWorks

HSMWorks is a CAM add on tool for SolidWorks. It is designed to generate the smoothest tool paths possible resulting in reduced machining time, improved surface finish and less tool wear[26]. HSMWorks features advanced 3-dimensional tool path strategies like Adaptive Clearing. Figure 2.9 shows an Adaptive Clearing tool path.

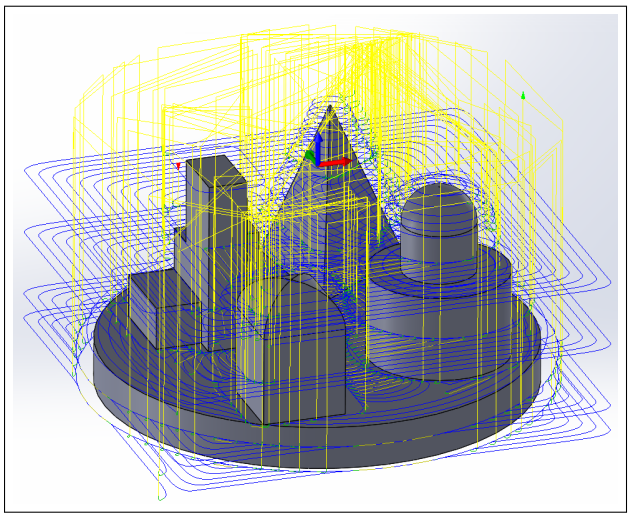

Figure 2.9: A screenshot of the Adaptive Clearing tool path generated by HSMWorks.

In this thesis HSMWorks has been used as a source of inspiration for tool path strategies, and to see what high end CAM software is capable of. However, being as intuitive and simple as a program of this calibre can possibly get, it is still challenging from time to time finding the correct ways to proceed for generating the tool paths needed.

2.5.6 The milling machines

Both milling machines used while working on the thesis are Do It Yourself(DIY) CNC milling machines built at the ROBIN group at the Institute for Informatics (ifi) at UiO. Both utilize Grbl compatible CNC controllers.

MidiMill

The MidiMill is a 3-axis milling machine used in this thesis to mill foam. Figure 2.10 on the next page show the machine.

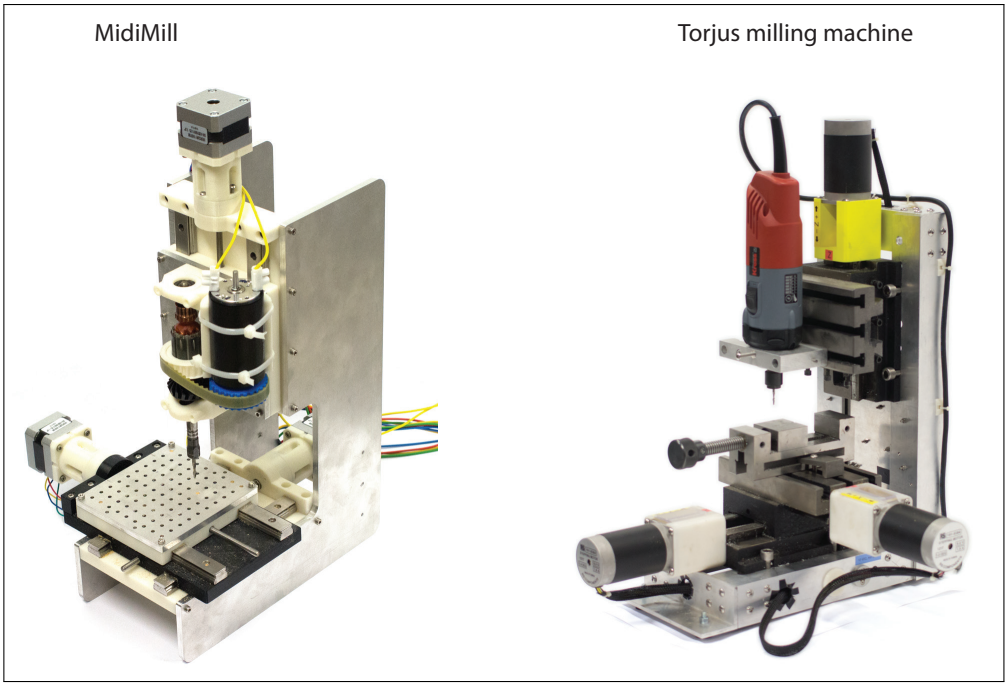

Figure 2.10: The CNC milling machines used in the thesis[45].

Torjus's Milling machine

This is another 3-axis milling machine built by Torjus Spilling for his master thesis *Self-Improving CNC Milling Machine*[45]. Having stronger stepper motors and a more powerful spindle than the MidiMill, it is capable of milling metals. In this thesis it has been used to mill aluminium.

Part II

The project

Chapter 3

Planning the project

How will this system be any different from the other CAM software out there that promises easy use and automated tool path generation? Well, simplicity needs to be taken to the next level by keeping the user inputs and interactions to the absolute minimum. To achieve this however, some constraints has to be applied.

3.1 This thesis

For this thesis, the machine used will be the DIY in house milling machines; the MidiMill and Torjus milling machine. The implementation will use the following guidelines[24]:

- Only use stock material of a particular size, but can be different materials.
- Only mill from the top surface of the material.
- Only use one or a few milling strategies.
- Only use one type of mill tool/cutter.
- Program loads STL(CAD) files and gives the user the ability to press "print" and the machine and software does the rest.
- Software has to make collisions *impossible*.

3.1.1 How to mill

A part can be imagined stuck inside the stock material and milling is a way to free it. There are many ways to go. One can for example mill away all the material above the part as Figure 3.1 on the following page illustrates, but then there is the problem with work holding when reaching the end of the milling operation. Even if one could find a way to get the whole part loose

and away from the stock, doing it this way will always leave the parts flat underneath. This limits the variety of parts that can be milled drastically. Another way to go is milling out a pocket around the part, leaving a frame

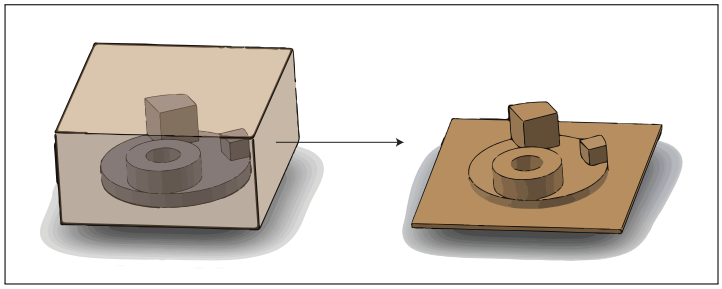

Figure 3.1: Illustration of how a part could be extracted from the stock material.

of stock material around it. This would solve the work holding problem since the holder now has something to hold on to. In addition it would take less time because less material has to be removed. However, there is still the issue of the parts having to be flat underneath.

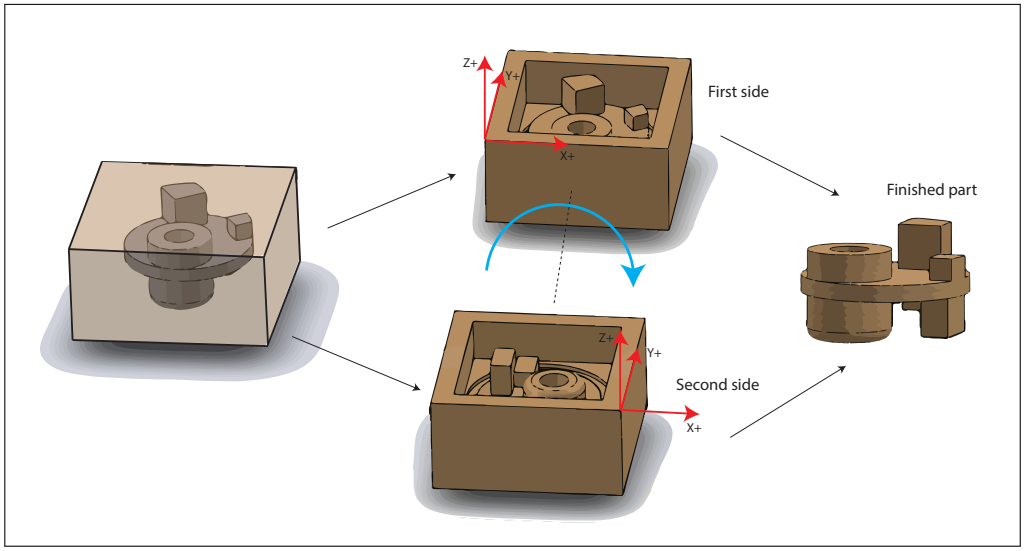

Figure 3.2: Illustration of how a part is extracted from the stock material in this thesis.

In this thesis the solution became milling a pocket, but only approximately halfway down the stock. The user then turns the stock around the y-axis when the first side is done. Another pocket is then milled on the other side of the stock. Figure 3.2 shows the process. When the stock is flipped, the origin of the work coordinate system is moved to the other side because the coordinates will be mirrored. A thin layer of material is left around the finished part, working as a sacrificial support to keep it in

place. The user can then break free the part from the stock with little effort. Now, the parts can be fully 3 dimensional, but there are still some restrictions to what a part can look like.

3.1.2 Work-holding

An important thing that keeps CNC milling from being rapid prototyping is work-holding. As mentioned, the vice is a commonly used work holder. Figure 3.3 shows a typical vice used for milling. It is not uncommon that the work piece needs to be repositioned during a milling operation. When using a vice it can be time consuming to reattach the work piece correctly with reference to the work coordinate system made up by the vice shown in Figure 3.3. Since the stock material is of a set size, a different work holder

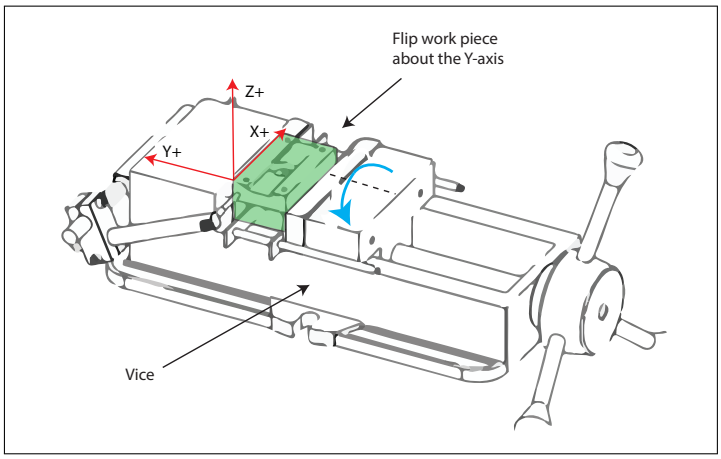

Figure 3.3: Illustration of a vice and the work coordinate system made up by it.

than a vice can be used. Some criteria the work holder should meet:

- It has to be easy to place and remove the stock from the holder.
- The stock should be fixed inside the holder without freedom to move.
- When fitting new stock material, one should not have to think about aligning it with the work coordinate system introduced by the work holder.

For this thesis the solution became a work holder that is sort of like a slot the stock slides into as can be seen in Figure 3.4 on the following page.

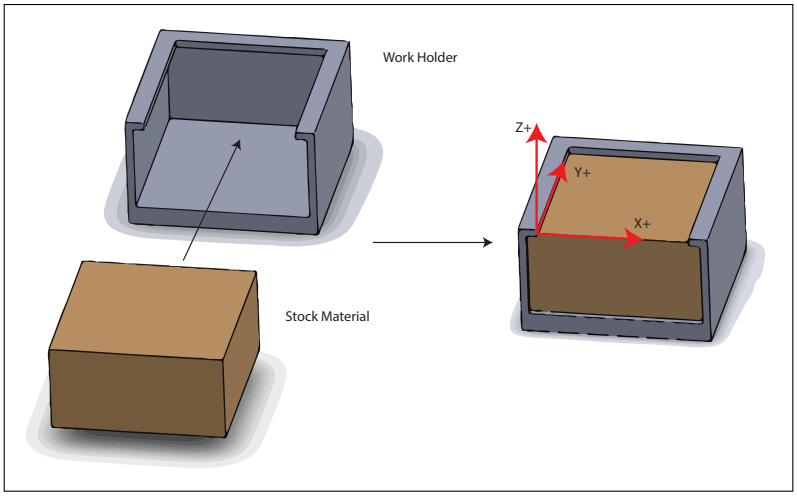

Figure 3.4: Concept of the work holder for this thesis.

3.2 Milling strategies

In this thesis, 2.5D milling is used for both roughing and finishing processes. This means milling on layer at a time. The majority of CNC milling tasks can be performed using 2.5D milling, due to the fact that a surprisingly large number of mechanical parts are 2.5D[32]. When milling, the endmill traces a sequence of coordinates to approximate the area that is to be milled. The pattern of tracing is called tool path topology. Path topology and the method used to link generated paths directly affect the machining time[33].

3.2.1 Roughing strategies

For milling freeform areas like pockets, contour parallel and direction parallel are most widely used[33]. In this thesis some variations of these are tried out.

Contour parallel (Offset) tool path

A popular path style in pocket machining is called contour parallel path, and is generated by successive offsets of the input boundary[32]. Figure 3.5 on the next page shows an example of what a basic contour parallel tool path might look like. When using a contour parallel tool path, the tool will have to accelerate and decelerate because the tool path may easily consist of many small path segments. This can increase the overall milling time[31].

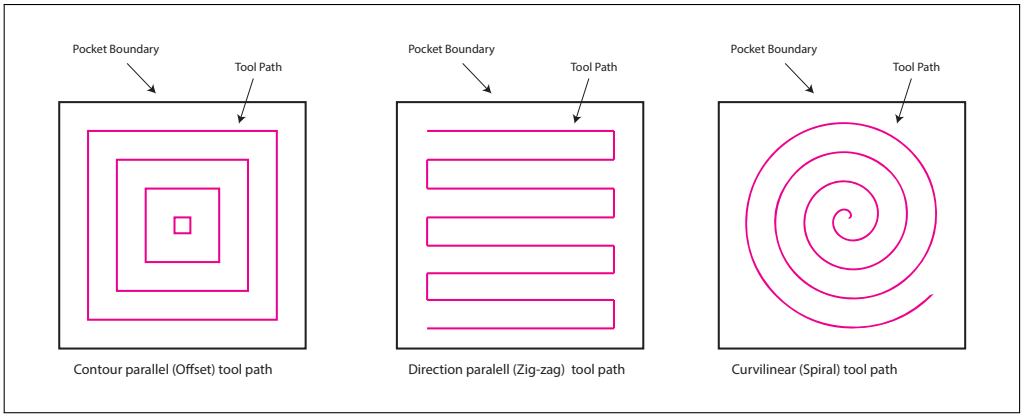

Figure 3.5: Simple illustrations of the different tool paths used in the thesis.

Direction parallel (Zig-zag) tool path

The idea of direction parallel milling, known as zigzag milling, is simple. The tool path consists of line segments parallel to each other, which the end mill follows alternating moving from left to right[17]. This can be seen in Figure 3.5. An advantage with the zigzag tool path is that it can be easily visualized and that it has the ability to maintain constant chip loads[31]. A disadvantage however, is that since the end mill moves back and forth, it will mill both with and against the cutter rotation.

Curvilinear (Spiral) tool path

In a curvilinear path, the tool travels along a gradually growing spiral. The spiral starts in the centre of the pocket and moves outwards towards the pocket boundary as Figure 3.5 illustrates. An advantage using a curvilinear tool path is reduced wear on the tool because the direction of the path changes progressively and local acceleration and deceleration of the tool are minimized[51]. In addition, like the zig-zag tool path, constant chip loads can be maintained.

3.2.2 Finishing strategies

The finishing tool paths used in this thesis are very simple. They are inspired by something called waterline finishing. If the part was to be emerged from a pool of water, the tool paths would be the lines that surround the contour boundaries of the part as it ascends. These are also known as constant z tool paths. Figure 3.6 on the following page illustrates this. It works very well for steep walls, but when the angles decrease its efficiency varies[5]. Therefore very small step-downs are used. Step-down is the distance between slices of the part, i.e. the part is emerged from

the water in very small increments. This way the surface finish will still be satisfying.

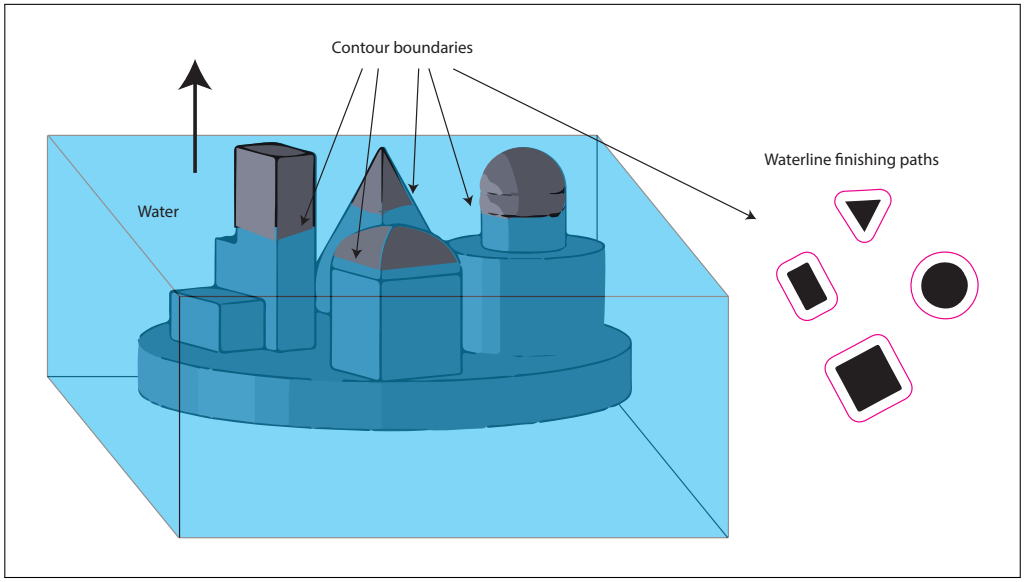

Figure 3.6: Figure illustrating waterline finishing. Imagine the part rising up from the water.

3.2.3 Tools

Changing the tool during a milling operation is not very user friendly and definitely not rapid prototyping. E.g. after changing the tool the position in the z-axis has to be reset which can be a hassle. Because of this, only one tool is used for the purpose of this thesis, the flat mill. A flat mill is preferable because it does not leave scallops while milling flat surfaces and it can be used to plunge short distances into the stock material. Of course different diameters of the flat mill can be used, but the chosen diameter has to be used during the whole milling operation. Figure 3.7 shows an image of standard flat mill with two flutes.

Figure 3.7: Image of a flat mill like the ones used in the thesis[27].

3.2.4 Milling direction

There are two directions to mill; conventional milling and climb milling, also known as up and down milling. Both directions essentially produce the same result, but the dynamics and stability properties are not the same[28]. During conventional milling, the thickness of the chip starts at zero and increases until it reaches its maximum as seen in Figure 3.8. Since the cut is so light at the beginning, it does not actually cut the stock, just slide across. When enough pressure is built up, the end mill suddenly cut into the material. This sliding and sudden cutting can deform the material, leaving behind a poor finish[41].

When climb milling on the other hand, the teeth of the cutter hits the material at a definite point shown in Figure 3.8. Hence the chip produced will start at maximum width and decrease until they are disposed behind the cutter. The tooth of the cutter does not rub against the material, leaving a better surface finish than conventional milling. However climb milling can apply larger loads to the machine[41].

Since climb milling leaves a better surface finish, the contour parallel and curvilinear tool paths in this thesis will be generated to achieve as much climb milling as possible. The finishing paths will also take advantage of climb milling.

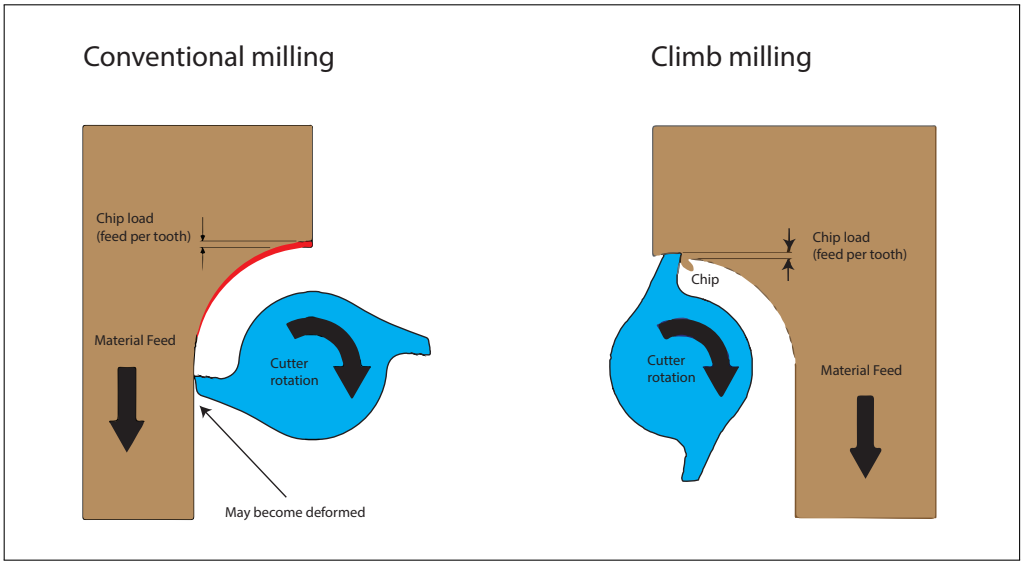

Figure 3.8: Illustrating the difference between conventional milling and climb milling[41].

Chapter 4

Implementation

This chapter shows how the software part of the system is implemented. From how STL files are read and handled. To how the models in the files are sliced into layers, and how the information in these layers is sorted and processed for later use.

Next it shows how the models are oriented to fit inside the stock material. Then how layers for use in the roughing and finishing operations are located and how the tool paths for both the roughing and finishing layers are calculated and how these are written to g-code files. In the end it takes a look at how the program created is merged with the software that transmit the g-code to the controller on the CNC milling machine.

4.1 Reading the STL files

First off is which STL files are accepted and how these are read.

4.1.1 Binary or ASCII?

As mentioned, an STL file is a description of a CAD model made up of a mesh of triangles. In the STL files, this description can be either binary or ASCII. The software made for this thesis only accepts binary STL files. Since binary representations take up less space, it greatly reduces the file size compared to ASCII. Beyond this, there are no further differences making one preferable over the other. This makes binary STL files the more popular choice.

4.1.2 Interpreting the file

When reading a STL file from the top down, the first thing it contains is the name of the model it is representing, called a header. This is 80 characters long and is usually ignored. When the file is read, the first 80 bytes of the file is skipped. Next is a 4 byte unsigned integer indicating the number of

triangles. Since it is little endian, the least significant byte is stored in the smallest address. For the program to read the number correctly, the order of the bytes has to be rearranged. After this the triangles are listed. As seen in figure 4.1 each triangle is described by 12 32-bit floating point numbers; the x, y and z coordinate of the normal vector and the three vertices of the triangle. These are also little endian and are flipped by bit shifting to get big endian. In total, 50 bytes are allocated to represent each triangle, but only 48 contain useful data. The last two bytes are called the attribute byte count and is not used. All the triangles are stored as a class Triangle and put in an ArrayList<Triangle> for further processing.

Bytes	Data type	Description
80	ASCII	Header. No data significance.
4	unsigned long integer	Number of facets in file
4	float	*i* for normal
4	float	*j*
4	float	*k*
4	float	*x* for vertex 1
4	float	*y*
4	float	*z*
4	float	*x* for vertex 2
4	float	*y*
4	float	*z*
4	float	*x* for vertex 3
4	float	*y*
4	float	*z*
2	unsigned integer	Attribute byte count

Figure 4.1: The syntax for a binary STL file

4.2 Slicing the model

In order to obtain information about the contour of the model in the STL file, the triangle mesh needs to be sliced. There are many ways to do this. Rodrigo M.M.H Gregori proposes an asymptotically optimal algorithm to slice a triangle mesh in a shortest possible run time[16]. However, to keep things simple which is an important part of the thesis, the method used is one borrowed from the world of video games, where triangles are cut by a plane to create three new triangles[30]. The reason for choosing this method is because the points located can be manipulated in any way needed for further use in the thesis. First the layers that will act as the planes slicing the mesh are created.

4.2.1 Making layers

The number of layers is calculated by simply starting with a value equal to half the total height of the stock material, then subtract the layer thickness until this value reaches negative half of the total stock height. For each subtraction an object of class Layer is created with the Z-value for this particular layer. The layers are then put in an ArrayList<Layer>.

Choosing a layer thickness

The layer thickness chosen for this thesis is 0.1 mm, or rather 0.1001 mm. This is because models are assumed created with different dimensions rounded up to the nearest whole mm.

4.2.2 Finding contours

The contours describe the boundary outlines of the model in a layer. To find the contours, the Z-value of this layer becomes a plane which slices the triangles of the triangle mesh as seen in figure 4.2. When a plane

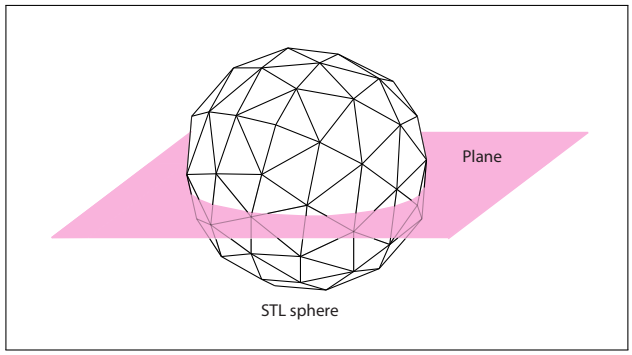

Figure 4.2: Triangle mesh sliced by a plane

slices a triangle you get two intersection points between the plane and the triangle shown in figure 4.3 on the following page. The line between these points makes up one line segment of a contour. The algorithm used to find the intersection points, is one originally used to split a triangle into three new triangles. This is called triangle clipping. For the purpose of the software for this thesis, only the two intersection points are needed, so the algorithm has been adjusted accordingly[30]. The algorithm is passed a triangle, the plane normal and a point on the plane. First it checks if the triangle is actually cut by the plane. A boolean variable for each of the three vertices is determined to be either true if on the positive side of the plane, or false if on the negative. If all are true or false, the triangle is not hit by the plane. But if e.g. one vertex is one the positive side while the other two are on the negative side, the triangle is hit. There are three points

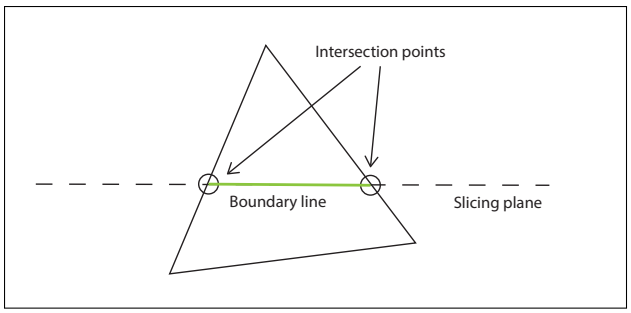

Figure 4.3: Intersection points between triangle and plane

representing the vertices; pA, pB and pC. Point pA is on the positive side of the plane, pB and pC on the negative. For each side of the triangle there are two points, e.g. pA and pB. These two points make up a ray, which is $vRay = pB - pA$ shown in figure 4.4. The fist thing to calculate is

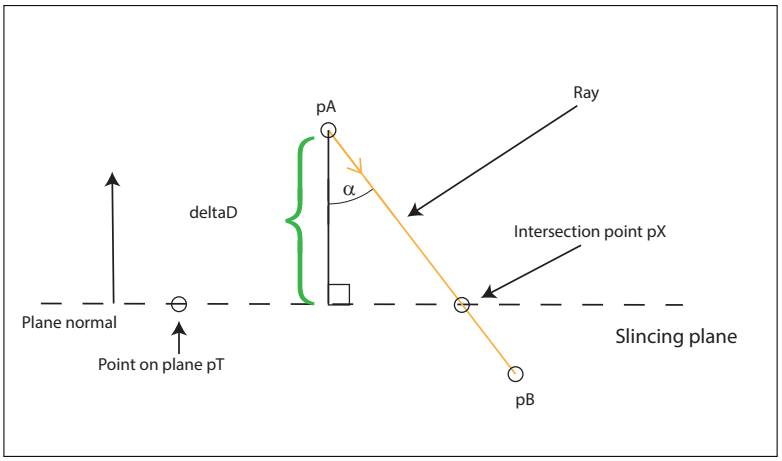

Figure 4.4: Ray intersection with plane

angle α, the angle between the ray and the plane normal. $cos\alpha$ is found by calculating the dot product between the normalized ray from pA to pB and the normalized plane normal;

$$cos\alpha = vnRayAtoB \cdot vnPlaneNormal$$

Next deltaD is found, this is the distance from point pA to the plane as seen in figure 4.4.

$$deltaD = (vPointOnPlane \cdot vnPlaneNormal) - (pA \cdot vnPlaneNormal)$$

Now that both $cos\alpha$ and $deltaD$ is known, finding the length to the intersection point pX from pA is done by simply finding the hypotenuse

of the right angle triangle shown in figure 4.4 on the preceding page.

$$length = \frac{deltaD}{cos\alpha}$$

The point pX is then calculated by scaling the ray vector from pA to pB.

$$pX = vnRayAtoB * length$$

The other intersection point is then found in the same way by using pC instead of pB. Once all the intersection points in a layer have been located, the boundary lines can be used to reproduce the outline of the figure in that layer.

4.2.3 Sorting intersection points

There is a problem that arises when traversing the list of triangles to find boundary lines. The order in which these lines are found is partly random. Figure 4.5 shows an example of how the order of the boundary line segments can occur, and what happens if traced in this order. For the

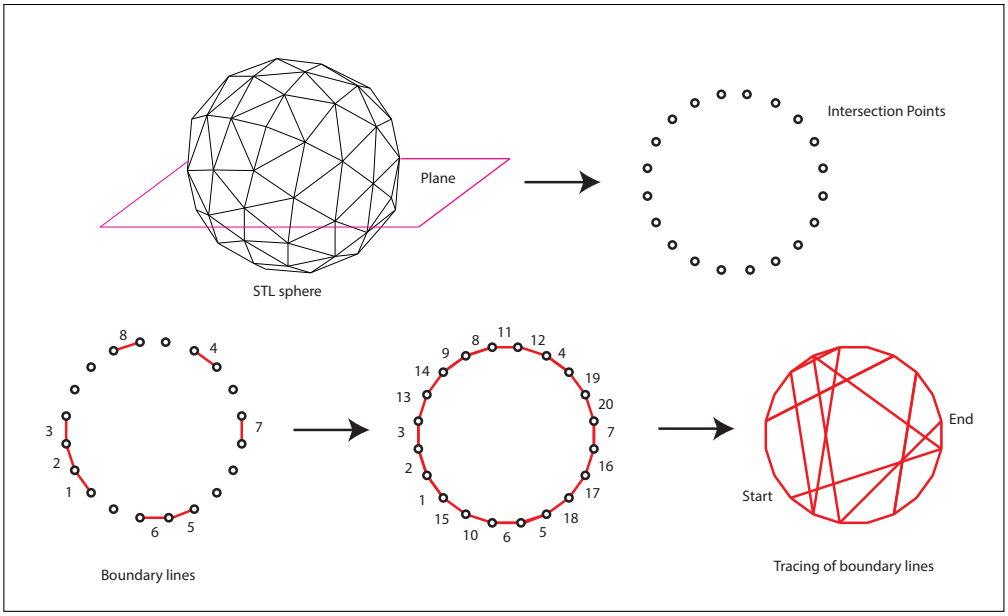

Figure 4.5: Occurrence of boundary lines

software to be able to recreate the shape that the boundary lines produce, all the lines have to be sorted. The intersection points must be stored in an array in a way that when traversing all the points, the starting point and end point is the same. Since every triangle produce two intersection points, each boundary line share an intersection point with two other boundary lines as figure 4.6 on the next page show. In order to sort them,

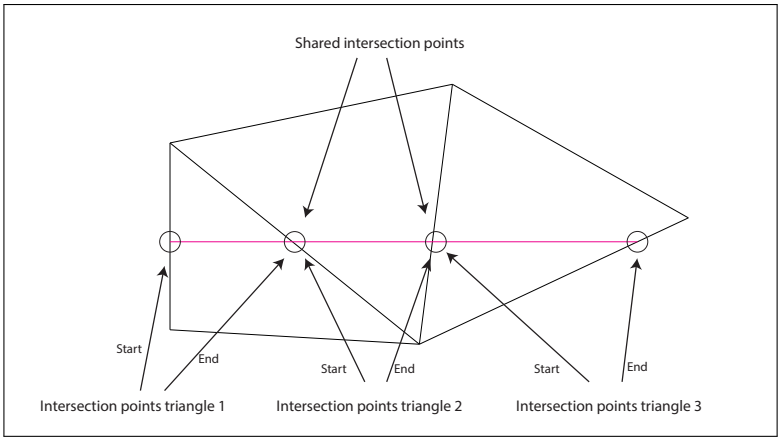

Figure 4.6: Sorting boundary lines

the end intersection point of the first line has to be identical to the start intersection point of the second line, and so on, all the way around until it reaches the start point of the first line. If the figure is complex enough, each layer can contain thousands of line segments that need to be sorted correctly. If done inefficiently, it will greatly increase the loading speed of the program.

Solution 1; brute force

One approach, and the first used in the software, is a brute force algorithm with some optimization. First, find all triangles that intercept the layer plane and put them in an ArrayList. Pick a random triangle. The line segment from this triangle is the starting one, and is removed from the list. Then go through the list until a triangle whose line segment's starting point, or ending point, is the same as the previous triangles end point. Remove this triangle and traverse the list again to find the next point. This is done until one of two scenarios occur. First scenario is if the number of sorted points equals the number of intersection points in that layer. If this happens, all the contours in this layer has been found. Second is if no triangle produce the intersection point we're looking for. When this happens, one of several contours has been found. The starting point for the next contour is chosen randomly from the list of triangles and the process is repeated.

Since the list of triangles is traversed for every point to be sorted, there is no denying this is a brute force algorithm. For simple figures that consist of only a few triangles, this approach is quite efficient. However as the figures become more complex and the number of triangles increase, this algorithm becomes very weak and time consuming. A solution where going through the list for every point can be avoided would be much faster. One way to do this is by using HashMaps.

Solution 2; HashMap

A HashMap is a class in Java that enables users to store a value with a key. The key can then be used to later retrieve the accompanying value. This is done with the simple operations put() and get(). For the purpose of this thesis it has been used to store and sort boundary lines. The slicing algorithm proposed by Gregori also make use of hashing to assemble the boundaries of contours in linear time[16].

When a boundary line is stored in the HashMap, the coordinate of the starting point is used as the key to retrieve it. Values stored in the HashMap are objects called MapVectors. Since there is no controlling which of the intersection points becomes the "correct" starting point, two MapVectors are created for each boundary line. They consist of two coordinates and a key. The two coordinates are the starting and ending point of a boundary line. The key, called *keyToOther*, is used to get a hold of the other MapVector created from that same line segment. Figure 4.7 shows how the two MapVectors are created and stored in the HashMap.

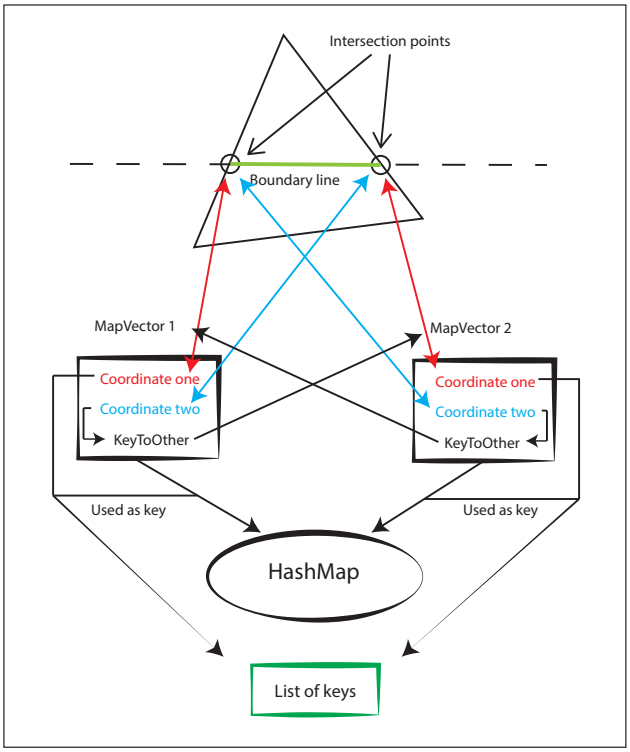

Figure 4.7: How MapVectors are created and stored in the HashMap.

Because all boundary lines share intersection points with other boundary lines as seen in figure 4.6 on the preceding page, there will be two MapVectors with the same starting point, hence the same key. Since all keys must be unique, a '1' is added to the end if the key is already being used in the HashMap. If the special case arises that even this key is in use,

the layer becomes marked as "corrupted". When creating keys, the values of the x and y coordinates is multiplied by 10 000 to separate points that are very close to each other. This is to avoid them being rounded to the same floating point value and get the same key.

For each layer the following procedure is carried out to sort the boundary lines. First, a random key is picked from the list of keys. This is used to retrieve a MapVector from the HashMap. Point one from this will be the starting point of a contour, and is stored in a sorted list. Then a loop is entered with the condition that the size of the HashMap is larger than zero. The second point of the MapVector is stored in the sorted list. The second point, coordinate two, is also the key to the next MapVector containing the next point needed. But before retrieving that, the current MapVector and the one that its *keyToOther* points to, is removed from the HashMap. Their keys are also removed from the list of keys. Not doing this will cause an infinite loop where the same MapVector is retrieved again and again. If the HashMap does not contain a value for the current key, a '1' is added to the end of it and it is checked again. In the case that no value is found after the second check and the size of the HashMap is larger than one, a whole contour has been stored in the sorted list of points. This is again stored in the corresponding layer object and the sorted list is cleared. A new key is extracted from the list of keys and the MapVector this produces will contain the first two points of the next contour in the layer. The procedure of sorting the rest is then repeated until either the size of the HashMap is zero or a new contour is found. Figure 4.8 shows the sorting process. If a layer is marked as "corrupted" it is skipped and

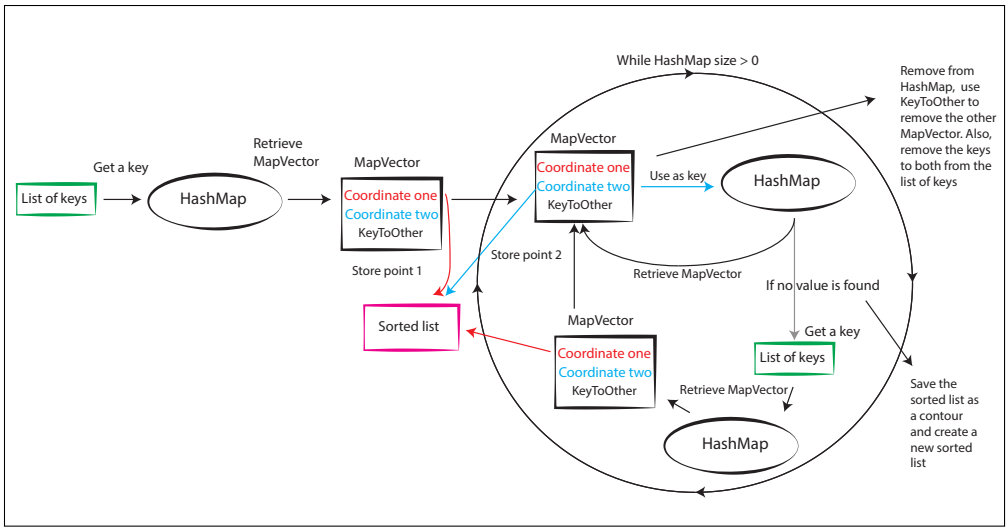

Figure 4.8: Structure of the HashMap implementation

will instead be a copy of the closest of the previous layers not corrupted. A corrupt layer may occur because two or more of the vertices of a triangle

lie on the same plane as the slicing layer. However this happens extremely seldom, and since the distance between layers is so small, skipping one layer is no crisis.

Figure 4.9 shows a comparison in speed between the two solutions. As expected, when the number of intersection points in the layers rises, the time used by the brute force approach grows exponentially while time used by the HashMap solution grow more linearly.

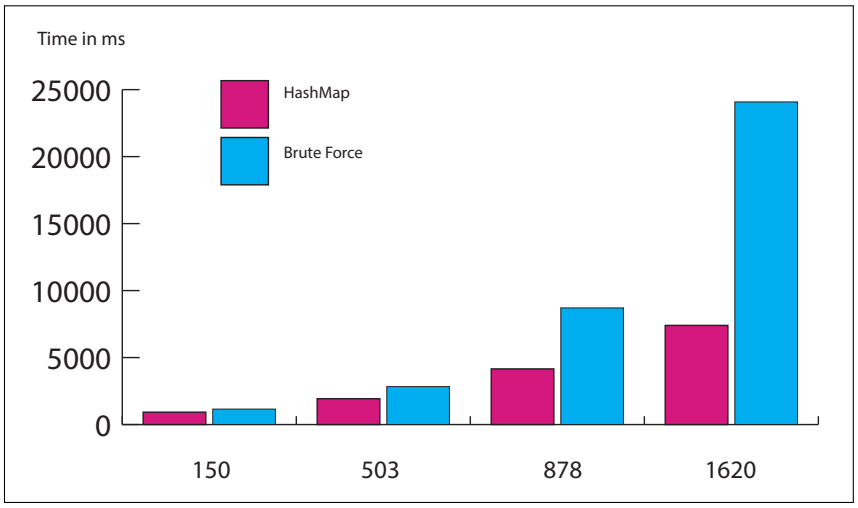

Figure 4.9: Time comparison between BruteForce and HashMap solution on four different figures. The numbers indicate the highest number of intersection points in a layer for each figure.

4.2.4 Representing contours

When all contours have been gathered and stored as sorted arrays of coordinates, they can be used to create Polygon objects. Polygon is a subclass of Geometry which is a class provided by the API JTS topology suite. Polygon objects are created by making a shell, which is a LinearRing created from the sorted coordinates. The reason for choosing JTS Geometry class to represent the contours is that it provides a whole range of useful methods to manipulate and compute new geometries shown later in this thesis.

4.3 The test program

The test program is an application made in Java to check the correctness of the generated contours as generated from the STL files. It is quite difficult to picture different shapes just by looking at an array of coordinates on a screen. This program aims to provide the needed visualisation.

4.3.1 Painting the contours

The application uses simple Java Graphics to paint the different layers with their contours. It consists of a run method with an infinite while loop. This calls the paint method every loop, then sleeps for a short amount of time before looping again. The contours are painted with the fillPolygon method. Different contours get different colours to easily identify and distinguish them. Figure 4.10 shows how the application displays different layers of a figure made in SolidWorks, saved as STL and loaded into the software. The white boundary around the contours is the maximum

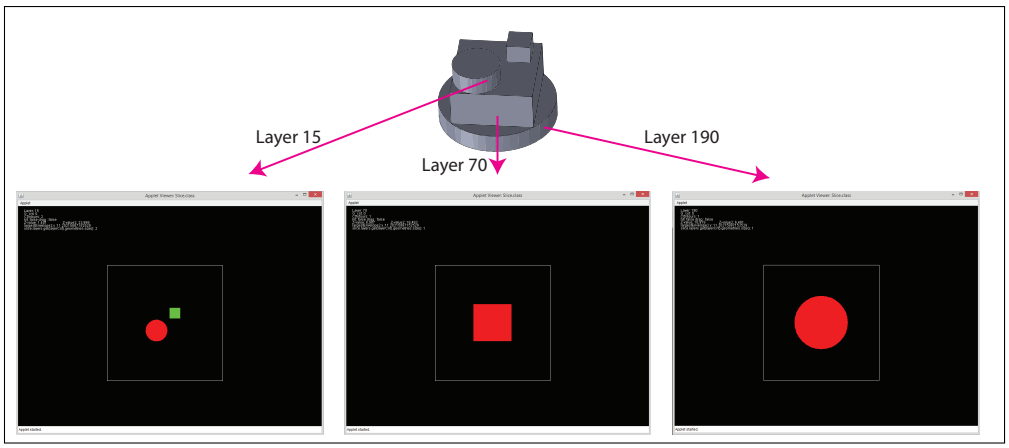

Figure 4.10: How the test application displays the different layers of a figure.

allowed size, i.e. the biggest pocket that can be milled. If contours are overlapping or are beyond this, the model cannot be milled. The run method also features easy switching of layers by using the arrow keys on the keyboard to switch back and forth.

4.3.2 Finding holes

If a model contains holes, the contours of these will initially be stored in the same way as all other contours. One way to find holes from a triangle mesh is using the normals provided in the STL file. These normals point out from the volume of the parts. To blindly rely on the face normals to decide what is the inside and the outside of the volume can be a bad idea. A problem that happens on occasion is that some of the normals are reversed. When milling, it could really mess up the model if it finds a hole that is not supposed to be there. In this thesis the face normals are not used. Instead, useful methods from the JTS Geometry class are used to determine which contours are holes and which are solid. To locate the holes, all contours found are iterated. The method *within()* is used to check if a contour is within another. If this is the case, and the contour within the other has a

smaller area, this is determined to be a hole. It is removed from the list of the other geometries, and put in a list of holes inside the layer class. When painting the contours, holes are coloured black to match the background as seen in figure 4.11. It may look like there are holes in the contour, but they are actually just black contours painted on top of the larger contour.

4.3.3 Positioning of models

To be able to mill pockets from both sides of the stock in a way that the two sides of the model matches, the model needs to be in the centre of the stock material. Both in the x-y plane and in the z-direction. To achieve this, the layer containing the middle of the model needs to be located. In this thesis this layer is called the middle layer.

The middle layer

The first thing the application does after reading the STL file and creating the layers, is going through them and making the sorted contour arrays into Polygon objects. The areas of these polygons are then measured. The middle layer, the layer that will be the final layer when milling from both sides, will be the layer that have only one contour, and this contour's area is the largest in the model. To get the area of a polygon, one simply uses the method getArea(). Figure 4.11 shows which layers will be chosen as the middle layer for three different parts. The visualizations from the application are mirrored compared to the actual parts.

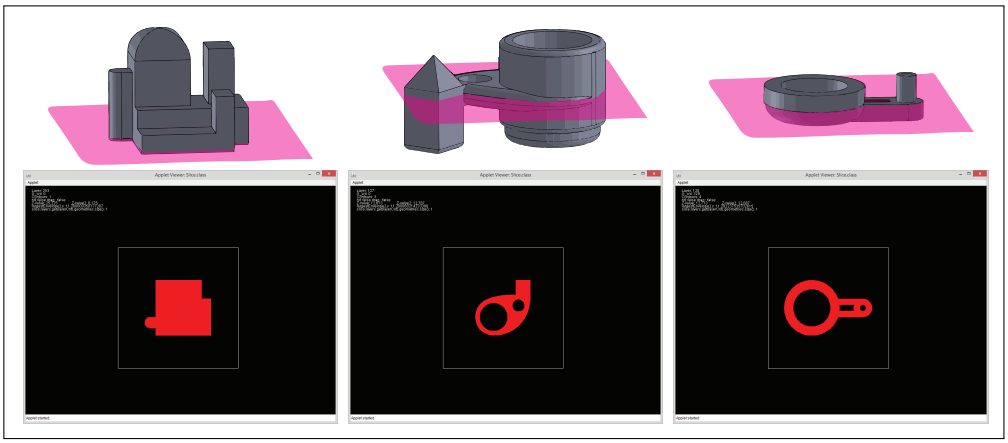

Figure 4.11: The different middle layers for different parts.

When creating the layers, the model is moved to the centre of the stock material by subtracting the maximum z-value of the model divided by two from the triangles vertices. Afterwards, when the middle layer has been

located, the process of creating layers is repeated. Only this time the z-value of the middle layer is passed as a parameter. This value is added to half the stock height, and is along with the maximum z-value of the model divided by two subtracted from the z-value of the triangles vertices. The model will now be placed in the stock material in a fashion so that the middle layer is in fact the middle layer of the stock material. Figure 4.12 shows this. There are instances when this is not possible, e.g. the middle layer is located in a way that moving it to the middle makes the model not fit in the stock. If this is the case, the model will be moved so it fits inside the stock whilst still keeping it as close to the middle as possible. This is achieved by simply adding or subtracting the amount the model is outside the stock, depending on if it protrudes from the top or the bottom.

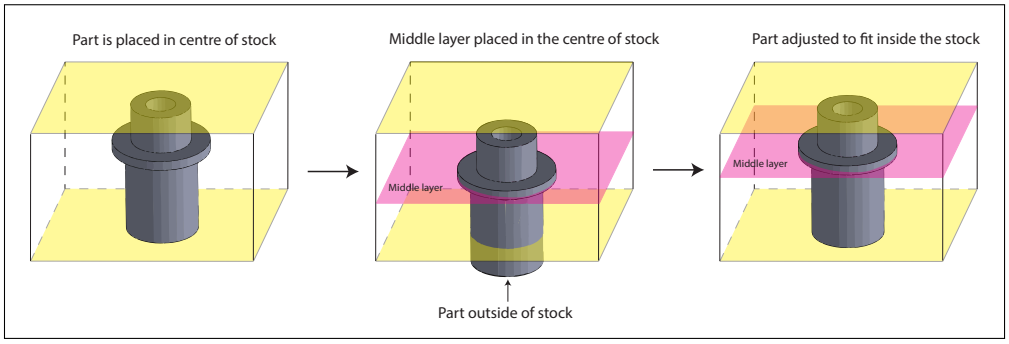

Figure 4.12: Shows the placement of the model in the stock material before and after a middle layer has been chosen

Orienting the model

When a model is loaded into the program, it could easily happen that the model is not oriented correctly as shown in figure 4.13 on the facing page. This could make the model impossible to mill. If the program determines that the model cannot be milled, it will go through the list of triangles from the STL file and swap the x and z coordinates. This will rotate the model 90°around the x-axis. Then the program will try loading the model again and see if it can be milled with this new orientation.

Centre the model

As seen in figure 4.11 on the previous page the contours are placed in the middle of the stock material. The largest contour in the model is also used to centre the model on the x-y plane. This is done by creating an envelope around that contour. The envelope will be the smallest rectangle that envelops the whole polygon that is the contour. Another polygon called limit is also created, this is a rectangle representing the absolute limits in

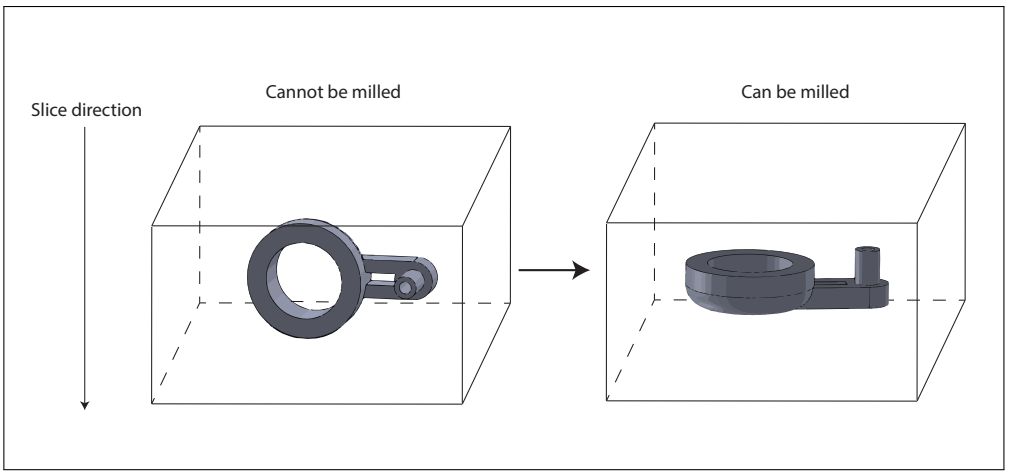

Figure 4.13: Shows an incorrectly oriented model

the x and y directions for the pocket in the stock material. If a contour goes outside this limit, it cannot be milled. The centre coordinate for both the envelope and the limit is found, and the differences in x and y between these are passed as parameters to a method setNewPos(). Here, the layers are iterated and the coordinates in the sorted lists are updated by adding the differences to x and y.

4.3.4 Finding roughing and finishing layers

The milling process consist of two operations; roughing and finishing. Because of this, two types of layers are defined in this thesis; roughing and finishing layers. Roughing layers are the layers containing roughing tool paths that are milled in the roughing part of the milling process. Finishing layers contain finishing paths that are milled in the finishing operation. This section shows how these layers are located.

Identifying contours

The contours are stored in an ArrayList<Geometry> inside the Layer class. The first contour found will be first in this list, second will be second and so on. The problem is that the first contour located in one layer, may be found second in the next. This is an unfortunate situation since it would be much more convenient if the same contours where located at the same array position in all the layers. This way, a contour can be compared to the same contour in the previous layer. When the contour Geometries are recreated in setNewPos(), this problem is addressed. An ArrayList<Geometry> called *outerContours* is created, this will be a list of all contours in a layer that is not determined to be holes. This list is the "correct" order in which the contours will be stored in all layers. When the layers are iterated, each

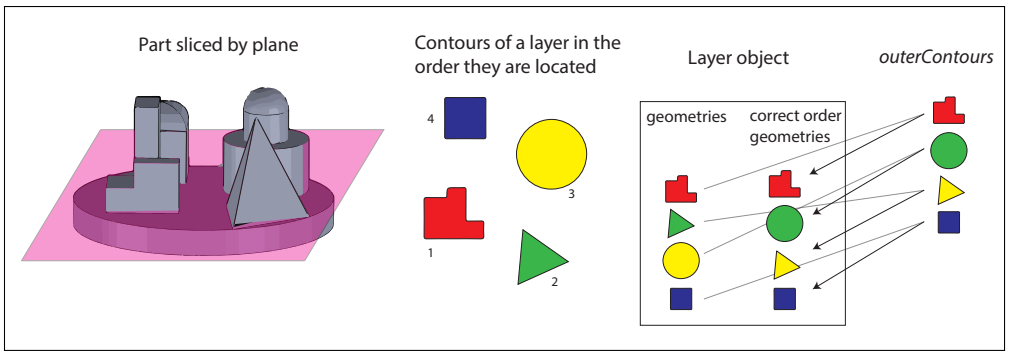

Figure 4.14: Figure illustrating how the contours of a layer is stored in a given sequence.

contour in a layer is compared to the contours in *outerContours* list. If a contour is decided to be the same as on in the list, it is stored in the same position as the contour in *outerContours* as Figure 4.14 illustrates. The position index in *outerContours* is then updated with the new contour. If a matching contour cannot be found in the list, the contour is added to *outerContours*. To decide if a contour is the same as another, the JTS bloolean methods touches() and within() are used. They return true if a polygon touches another and if a polygon is within another polygon. Since the hole contours have already been removed, if one or both of these return true, the contours are the same.

Roughing layers

The roughing layers are layers that are essential to keep the measurements of the model correct, and keeping the milling machine from destroying itself. These layers are stored in an ArrayList<Integer> *roughing*. There are three ways layers are added to this list:

- When going through the layers and a new contour is found, i.e. one not in the *outerContours* list, the layer before this one becomes a roughing layer. This is to keep the measurements in the z - direction as correct as possible. The first case in Figure 4.15 on the next page shows this condition.

- If a contour is considerably larger in the current layer than the previous one, the previous layer is added to *roughing*. This is shown in case two in Figure 4.15 on the facing page. This is also done to keep the measurements correct.

- The final way a layer is added to the list is by a method *fillRoughing()*. Since there is a limit in how deep an end mill can mill without breaking or deflecting, adding a sufficient amount of roughing layers

is vital. For aluminium the milling depth is set to 1 mm, this parameter is sent to *fillRoughing()*. It then fills in roughing layers around the already existing ones, so that there is at least one roughing layer every 1 mm.

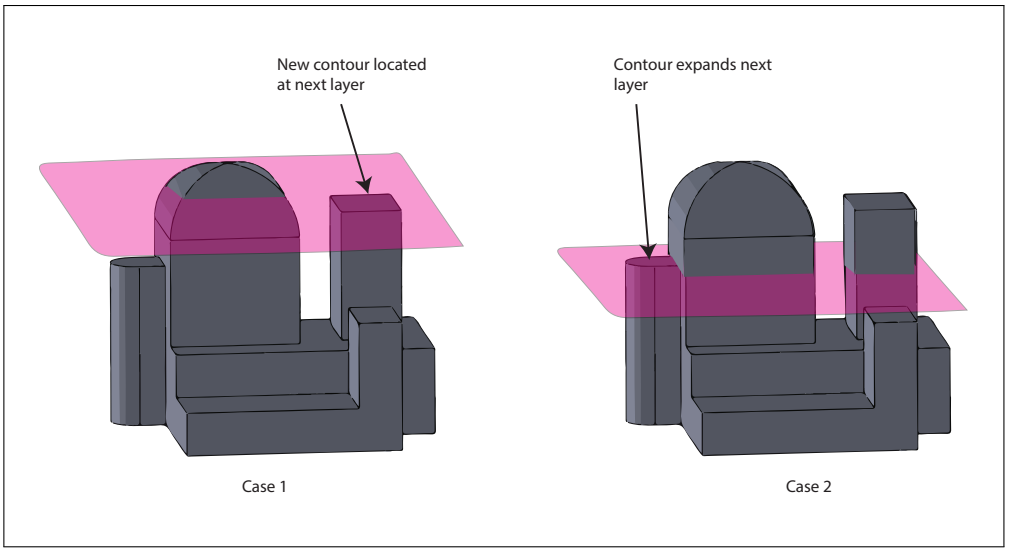

Figure 4.15: Shows two conditions for adding a roughing layer

Finishing layers

Finishing layers are the layers that provide the final surface finish when milling a part. A layer becomes a finishing layer when the area of one or more contours in the next layer is slightly bigger than the same contours in the current layer, i.e. when a contour changes.

The finishing layers are first stored in a two dimensional integer array called *finishing*. The first dimension indicates the contour that has changed and the second indicate which layer. Finishing layers are then transferred to an array of class *FinishingPaths* containing an ArrayList of the finishing path and what contour the path is for. If a layer qualifies to be a finishing layer, it is added to *finishing* if it is not already located in *roughing*. If the layer is the middle layer it will not be added. Figure 4.16 on the next page shows which layers of two parts will be finishing layers. If a contour changes from layer 20 through 30, then again at layer 100 through 120, these will be separated with an impossible value in the layer dimension of *finishing*. This is just to make sure the end mill does not go straight to layer 100 from 30, which could cause a collision with the model.

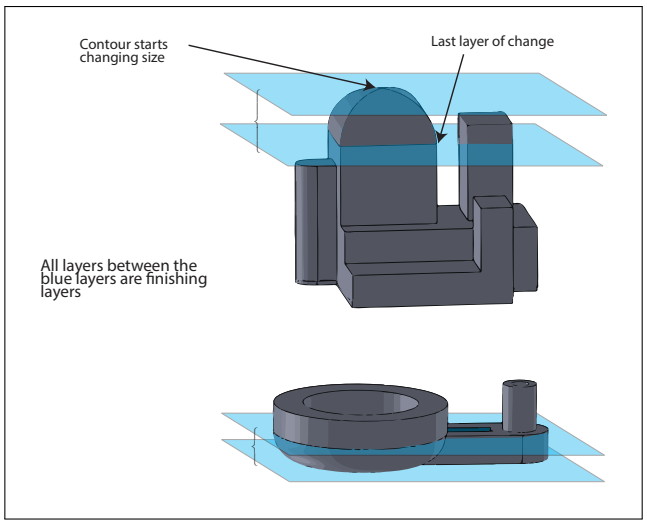

Figure 4.16: Shows the finishing layers of two different parts

4.4 Implementing roughing milling strategies

This section addresses the implementation of the roughing strategies decided upon in section 3.2.1. To make things easier, in this chapter and for the rest of the thesis, these will be referred to as the offset (contour parallel), zig-zag (direction parallel) and spiral (curvilinear) strategies. An additional variant of the spiral strategy has also been implemented. This is a crossover between the zig-zag and spiral strategy.

4.4.1 Offset milling

The offset milling strategy rely on the JTS Geometry method *buffer(double distance)*. This method computes a new Geometry with an offset a distance inward or outward from the original geometry depending on the distance parameter being negative or positive. Figure 4.17 on the facing page shows how it works.

First, the outer contours for a layer are traversed, i.e. not holes, only solid contours. Using the JTS method *symDifference()*, the symmetric difference is found between the largest envelope of the model and the contours of a layer. The result is a new single rectangular polygon geometry with holes where the original contours should have been. Figure 4.18 on page 52 shows how this may look. Then by creating a negative buffer from this polygon, one can extract both the exterior ring and interior rings of the new polygon shown in Figure 4.18 on page 52. This is then repeated with a bigger and bigger negative buffer until no polygons are produced. All the rings are in an *ArrayList<Geometry>* called *leftoverHoles*. The difference in negative value from one buffer

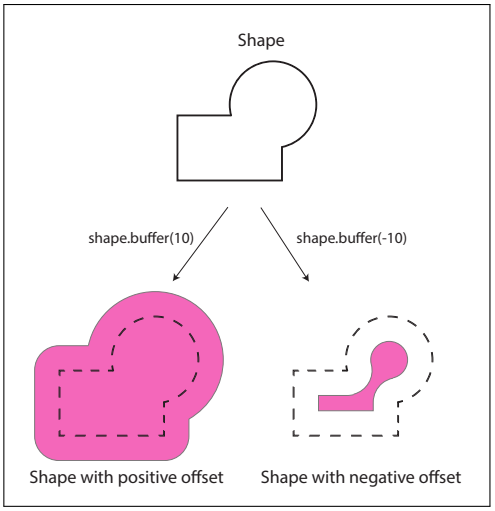

Figure 4.17: The JTS buffer method

operation to the next will be the step-over when milling. E.g. if the diameter of the end mill is 3 mm and desired step-over is 40%, the buffer value becomes $3 * 0.4 = 1.2$. The first buffer value however must be equal to half of the end mill diameter. This is to get correct dimensions of the contours when milling. For the 3 mm endmill and 40% step-over the first buffer value will be $3/2 = 1.5$, the second $1.5 + 1.2 = 2.7$, the third $1.5 + 1.2 + 1.2 = 3.9$ and so on. Remember all these values are negative.

Now comes the tricky part. All paths the end mill has to follow to mill the layer have been located. The problem is moving between these paths in such a way that the end mill:

- does not collide with the solid contours of the model.
- follows all paths only once, and ends up at the starting position.
- get a small amount of time milling with material on both sides when following a path. This is called slot milling and will happen from time to time.
- follows the paths in a way that climb milling is achieved, in particular when milling the final paths closest to the contours.

After all the polygon paths have been found and stored in *leftoverHoles*, a *while* loop with the condition $leftoverHoles > 0$ is entered. A tiny polygon is created at the starting point, which is the corner of the largest envelope. The list leftoverHoles is then traversed, and the polygon with the shortest distance to the starting polygon is located. Next, the arrays of coordinates representing the two paths around the two polygons are extracted. Using these arrays the closest coordinate from each polygon is found, i.e. where the two polygons are closest. The two coordinates is then

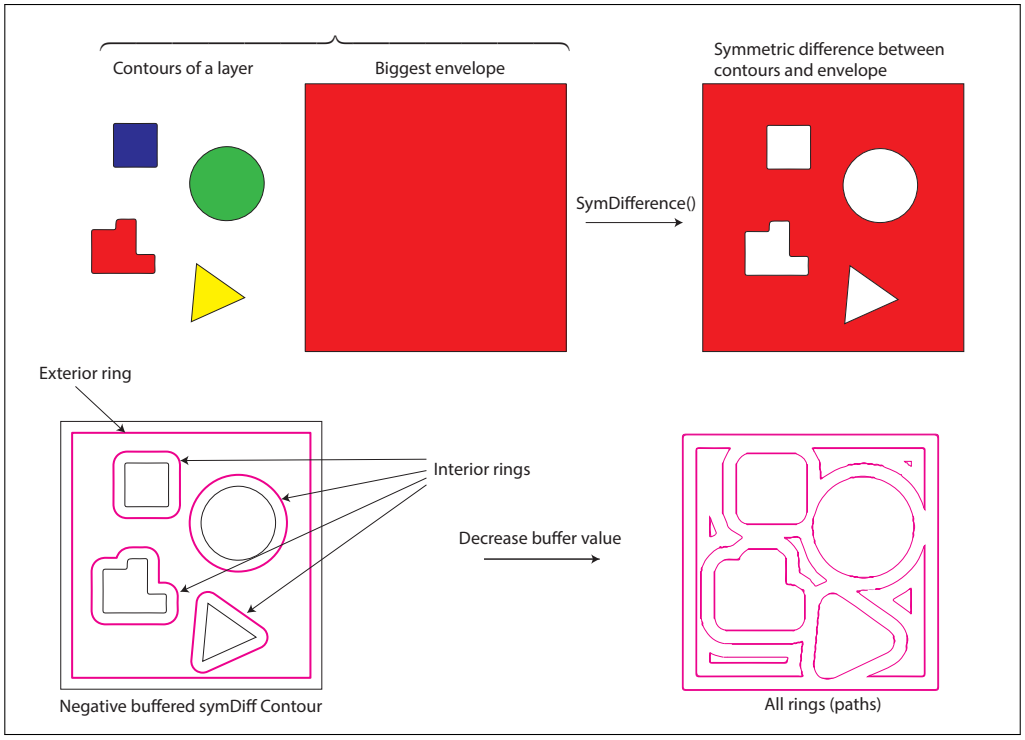

Figure 4.18: Finding offset paths

sent to a method *mergePolygons()* along with the path array so far, called *thePathCords*, and the coordinates of the second polygon. In *mergePolygons()* the *thePathCords* and the coordinates from the closest polygon are merged together at the two closest points. Figure 4.19 shows a simplified illustration of what this might look like using the paths from Figure 4.18. When a new path has been merged with *thePathCords*, it is removed from *leftoverHoles*. To reduce the amount of time milling with stock material on both sides of the end mill, the algorithm will look for paths with short distance to the last used closest points first. If the distance is too big, it will look for the path closest to *thePathCords* and merge with that.

To achieve climb milling, all polygons chosen to be merged is sent to a method *checkIfClockwise()*. This methods check if the polygons coordinate array is in a clockwise or anti-clockwise direction. The way it does that is by traversing the array, and for each coordinate it ads to a sum:

$$\sum_{i=1}^{antCords} (cords[i].x - cords[i-1].x) * (cords[i].y + cords[i-1].y) \quad (4.1)$$

If the sum is positive, the path around the polygon is clockwise, if negative it is anti-clockwise. When milling outer contours, the path has to go in the anti-clockwise direction to get climb milling. If the path already is anti-clockwise, nothing is done and the original polygon is returned. If not,

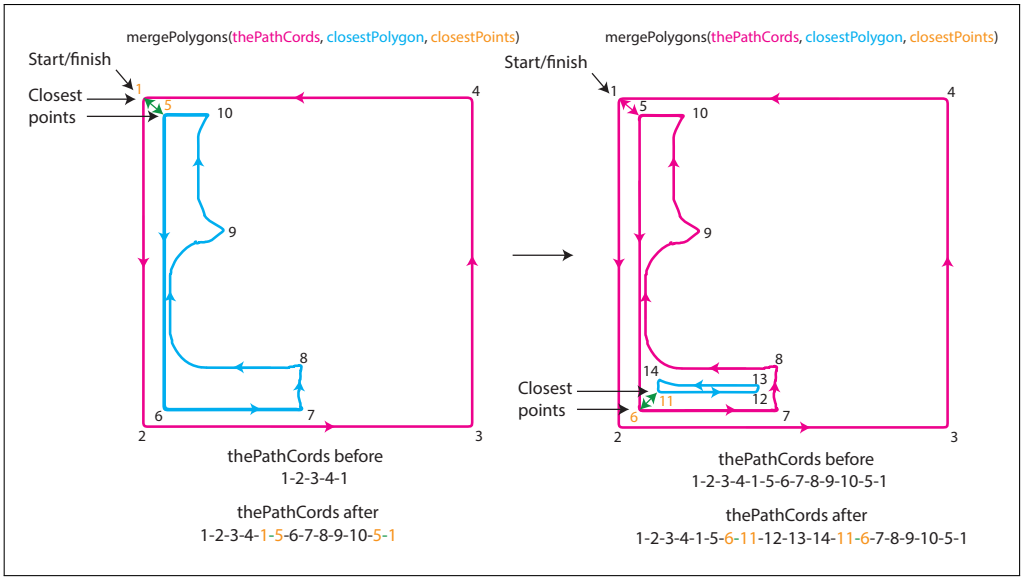

Figure 4.19: Simplified figure illustrating how paths are merged together.

a new polygon is created with the reversed path of the original polygon and this one is returned. If a layer is below the *middleLayer*, this process is flipped, i.e. clockwise paths are reversed. This is because the stock material is flipped around the y-axis, making the paths mirrored.

When *leftoverHoles* is empty, *thePathCords* is an array of coordinates that the end mill can follow without colliding with contours. The last coordinate in the array has the same value as the first. The array *thePathCords* is stored in the layer class. This algorithm only computes the paths for the layers in the *roughing* list, the other layers are skipped. Figure 4.20 on the following page shows a screenshot of a roughing layer using a 3 mm end mill with 40% step-over. The yellow line is the path between the last two coordinates, indicating the start/finish. The middle layer is a special case. When the path for this layer is calculated, only one buffer operation is performed. Since it only has one contour, the *positive* buffer to this contour is generated with buffer value of half the end mill diameter. The middle layers *thePathCords* only consist of the coordinate path around this one polygon.

4.4.2 Zig-zag milling

The zig-zag milling strategy is probably the most straight forward of the roughing strategies since the end mill just moves in a zig-zag pattern back and forth. The algorithm calculating these paths is thus pretty simple. First the lowest and highest x and y values are found in the largest envelope. These are the limits in which the path must stay inside, i.e. the pocket in the stock material. An empty array of coordinates called

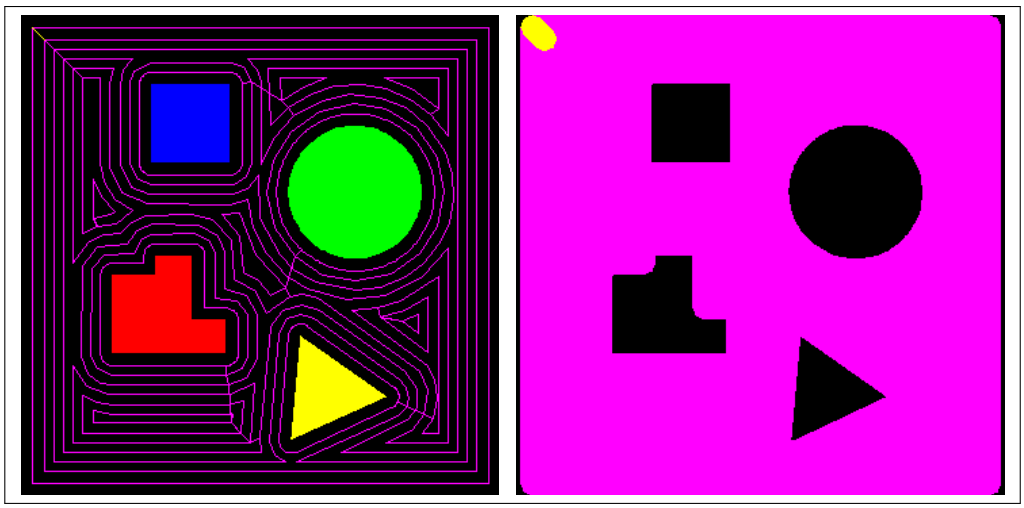

Figure 4.20: Screenshot of a roughing layer using the offset milling strategy.

thePath is created. The first coordinate is set to the top left corner of the largest envelope. Then, the next coordinate in the array is the one prior to it but with 0.5 mm subtracted from the y-component. This continues until the y-value of a coordinate exceeds the lower y-limit. This coordinate will instead increase its x-value with the step-over value. Then the next coordinates will increase their y-values with 0.5 mm until the upper y-limit is met. The algorithm continues this way, generating the path up and down until it reaches the maximum x-limit. A second path is also created. This one is the first path only reversed. When following the first path, then the second path, one ends up at the starting position of the first zig-zag path. When moving through the roughing layers, the paths alternate between the first and second path, making the end mill move back and forth.

For layers with no contours, the end mill can just follow the path without the possibility of crashing into anything. However, when layers do have contours something must be done. To check if the path collides with a contour, a positive buffer equal to half the tool diameter is made. If a coordinate in the path is inside this buffered contour, the position of this coordinate in *thePathCords* is stored in an array inside the layer class. This is an array of a class *MillHit*, which consist of the contour the path is inside and what position in the path it is in. A negative buffer is made for each hole if there is any. If a coordinate along the path is inside a holes buffer, that position is removed from the list. When the end mill reaches a coordinate in this list it will move to the layer just above the contour it is colliding with. It will move back down when it gets to a coordinate not in the list of colliding coordinates. Figure 4.21 on the next page shows a screenshot of a roughing layer using the zig-zag milling strategy with a 3mm end mill using 60% step-over. The grey lines indicate where the end mill moves above the contours. Compared to Figure 4.20 the zig-zag

strategy leaves behind a lot of excess stock material around the contours.

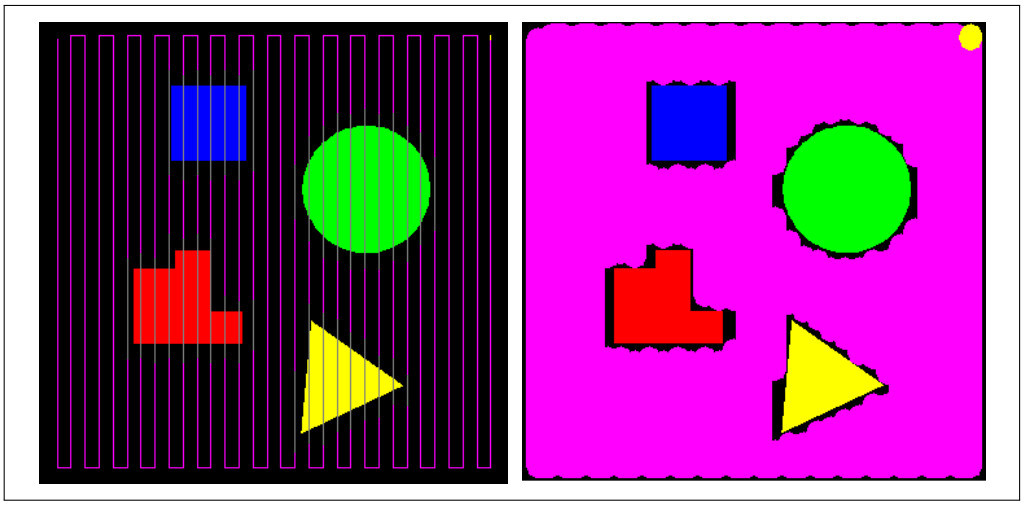

Figure 4.21: Screenshot of a roughing layer using the zig-zag milling strategy.

4.4.3 Spiral milling

The implementation of the spiral milling strategy is very similar to the zig-zag strategy, the only difference being the path is a spiral as opposed to in a zig-zag pattern. The spiral used is an Archimedean spiral. To achieve climb milling, the spiral grows in a counter-clockwise direction on layers above the middle layer and clockwise on layers below. The variable a determines which direction the spiral grows by being positive or negative. Theta is the angle in radians so for every 2π the spiral does a whole revolution. Variable a also determines the distance between each successive revolution as figure 4.22 on the following page shows.

The collision check with contours and holes is basically the same as in zig-zag except for one small thing. After finding which positions in the spiral path that collides with contours, another array is made. Only positions where it is safe to move is stored in this array. When coordinates from the spiral is inside a contour, these positions are removed from the array. This way when the spiral path is inside a contour, the end mill will move in a straight line over it instead of following the spiral path. This can be seen in figure 4.23 on page 57. Like the zig-zag strategy there is some excess stock left around the contours after the layer has been milled.

Corner spiral milling

The corner spiral roughing strategy can be thought of as a fusion between the zig-zag and spiral strategies. This is reflected in the implementation.

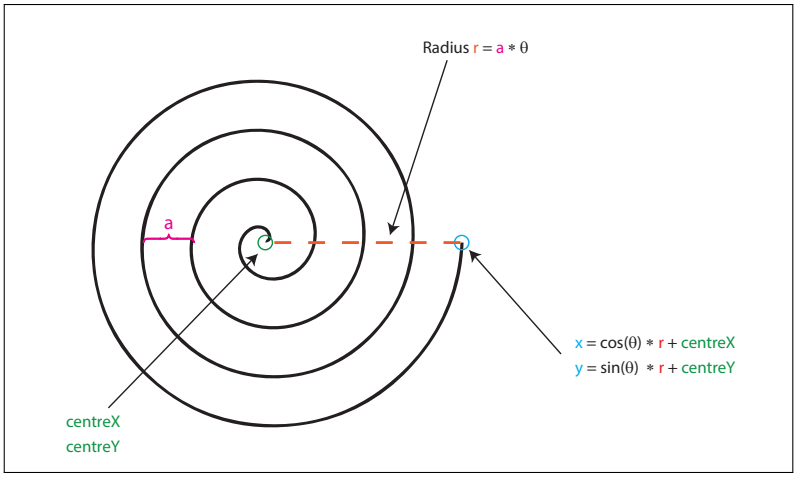

Figure 4.22: An arithmetic spiral, also known as Archimedean spiral.

It goes back and forth between two opposing corners while following the path of a growing spiral with its centre in the corners. As with zig-zag, multiple paths are created. However this time around the paths are not reversed copies of each other. In total four spiral paths are made. Two for the layers above the middle layer, and two for the ones below. As with the previous strategy, the spirals grow in a counter clockwise direction above, and clockwise direction below to get as much climb milling as possible. One path starts in the upper right corner of the largest envelope, and grows until a whole turning of the spiral is outside the envelope. Only coordinates in the spiral that is within the envelope is stored in the path. The other path starts in the bottom left corner as shown in figure 4.24 on page 58.

Avoiding collisions with contours is implemented in the same way as with regular spiral milling strategy. This means that there is left behind stock around contours as can be seen in figure 4.25 on page 59. To remove this excess material, all but the offset roughing strategy needs a "rough" finishing operation before moving on to the final finishing process.

4.4.4 Holes

Generating paths for the holes in a layer is done in almost the same way as when generating the offset roughing strategy paths. The algorithm first goes through the holes in the layer, and checks them up with a list of all the holes in the model. It then determines which hole this current hole is. Then buffers with negative buffer values are created from the hole and stored in a list over leftover contour paths. Then, just like with offset path generation these paths are merged together, only difference is that the path starts in the centre of the hole and works its way outwards. The holes path is then stored in the layer class in an array of a new class *Hole*, that consists

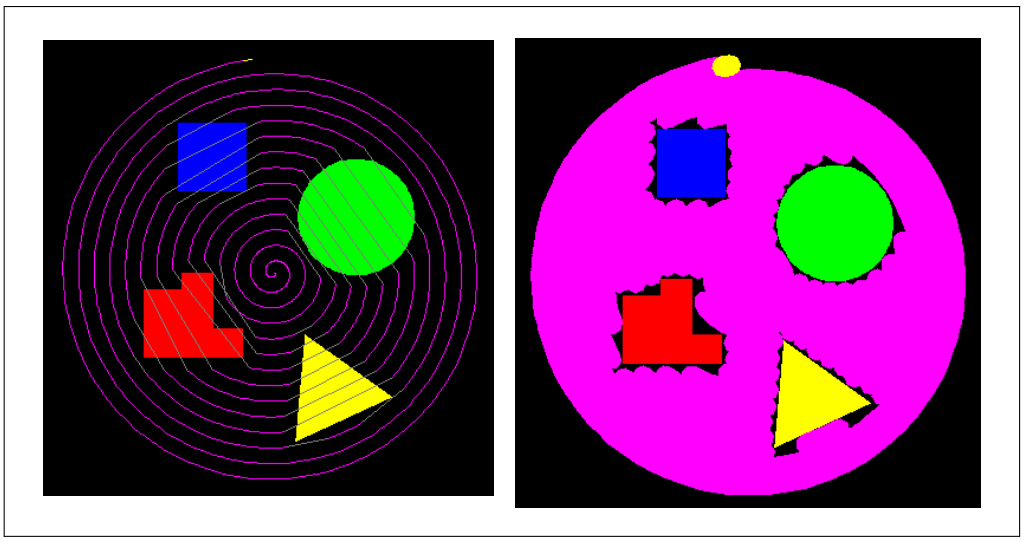

Figure 4.23: Screenshot of a roughing layer using the spiral milling strategy.

of list of the coordinate path. The same hole in different layers is stored under the same index in this array. For the zig-zag and spiral strategies, only one buffer is made since most of the material in the middle of the holes is already removed.

4.5 Implementing finishing milling strategies

The finishing operation in this thesis is divided into two parts; rough finish and final finish. Both utilizes the waterline finishing strategy decided upon in Section 3.3.2. The difference between them is the step-down distance.

4.5.1 Rough finish

Since all but one roughing strategy leaves behind a mess around contours, an initial finishing operation is required. The purpose of this process is to remove the excess stock around the contours of the model.

All roughing layers are traversed and also the array of contours within them. For each contour a positive buffer is created with a buffer value equal to half the diameter of the tool. This buffer geometry is then sent to *checkIfClockwise()* to get the coordinates oriented so that climb milling around the contour is achieved. The path of coordinates is then extracted from the geometry and stored in the layer class. Each contour's finish path is stored in an array with an index corresponding to its position in the correct geometry list in the layer shown in Figure 4.14 on page 48. This way, it is easy to traverse the roughing layers and the finishing paths one contour at the time when generating the g-code.

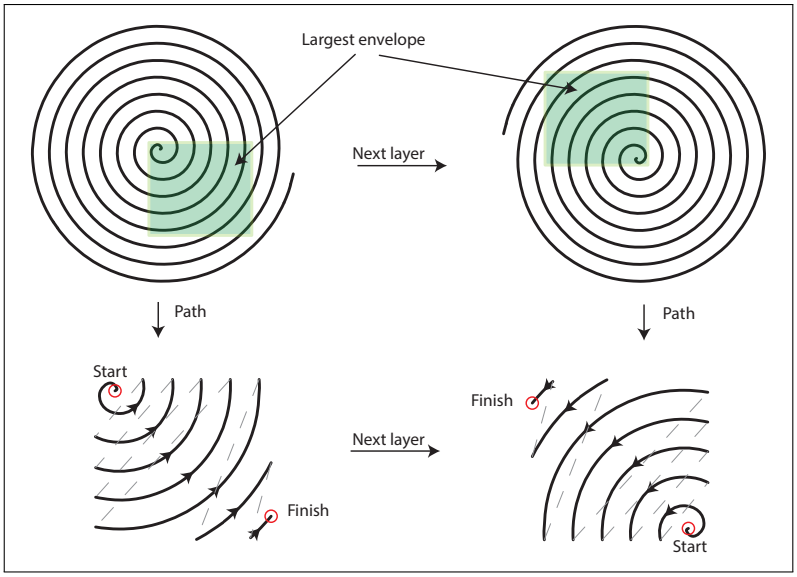

Figure 4.24: Generating corner spiral paths.

4.5.2 Final finishing

The final finishing process is needed if the model to be milled contains fillets or chamfer of some sort. Figure 4.16 on page 50 shows two parts that both require a final finishing operation. The model in figure 4.10 on page 44 on the other hand has no contours with gradual change, and has no need for this operation.

The layers that require this final finish are already stored in the two dimensional array *finishing*. Generating the finishing paths is done in basically the same way as with the rough finishing paths, only this time around, only contours containing layers in the *finishing* array will get a path. The paths are stored in a two dimensional Coordinate array inside each layer that has a contour that requires final finishing. The starting coordinate will be the coordinate in the finishing path that is closest to $x = 0$ and $y = 0$. This way, the starting position for the finishing path for a contour will be in almost the exact same spot when moving downwards through the layers. If the end mill had to move to a new starting position say on the other side of the contour, it would have to first move to the top of the stock before moving to the new position in order to avoid collision. This would take a colossal amount of time when there could easily be hundreds of finishing layers. In the test program, a finishing contour in a layer is indicated with a cyan path around it as figure 4.26 shows.

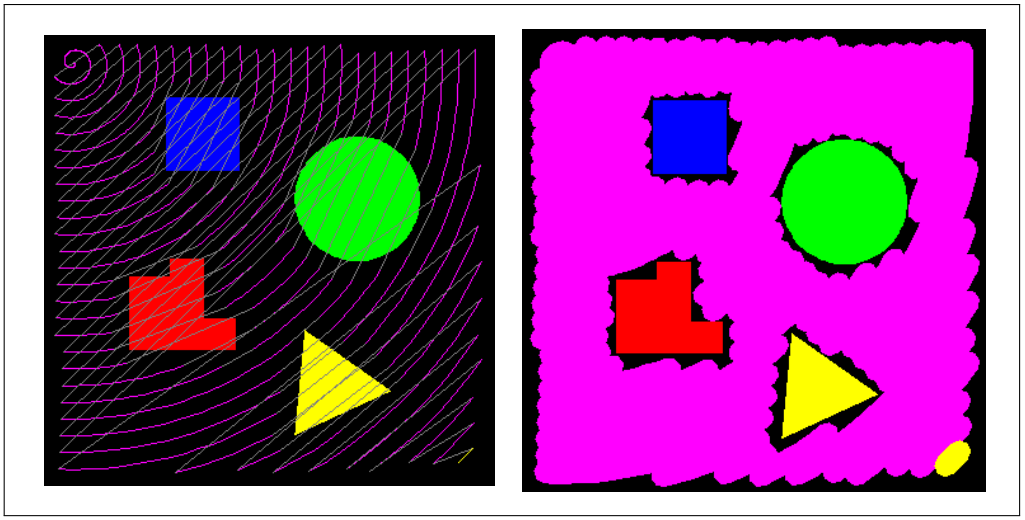

Figure 4.25: Screenshot of a roughing layer using the corner spiral milling strategy.

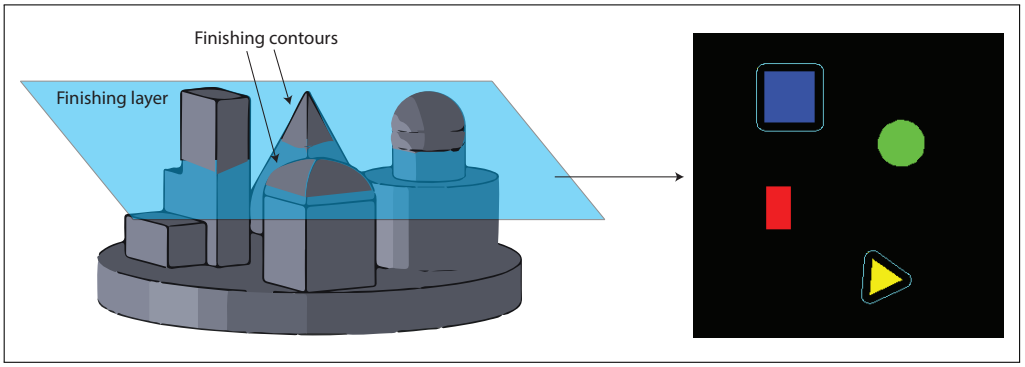

Figure 4.26: Screenshot of a finishing layer. The test program shows a mirrored image of the actual model.

4.6 Writing the g-code

For this thesis, to keep things simple, only linear motion, the G01 command, is used to describe the movement. The g-code is written to two separate files called *G_code.nc* and *G_code2.nc*. One file for the layers above the middle layer, and one for those below. The first line of code in these files is G90. This command tells the CNC controller to use absolute coordinates, i.e. positions are defined from reference to $x = 0$ $y = 0$ $z = 0$. The g-code is then written in the following order:

- Roughing paths.
- Hole paths.

- Rough finishing (if necessary).
- Final finish (if necessary).
- Cut out the middle layer.

4.6.1 Roughing paths

For the first g-code file, layers from zero to the middle layer are traversed, for the second file, layers below the middle layer are traversed in reversed order. This means starting at the last layer and going backwards towards the middle layer. If a layer is contained in the *roughing* list, its g-code path ArrayList of coordinates is iterated. For each coordinate in the list, the following line of code is written:

"G01 X"+(slice.layers.get(i).gCodePath.get(j).x)+
"Y"+(slice.layers.get(i).gCodePath.get(j).y)+""

For the second file the x-value set to be negative. This is because the origin of the work coordinate system is moved to the other side when the part is flipped about the y-axis as Figure 3.2 on page 28 illustrates. Next comes the z-value, this one is a bit more tricky. If the roughing strategy used is offset milling the z command is:

"G01 Z"+(-((totalStockHeight/2)-slice.layers.get(i).zValue))+""

Since the z-values range from $totalStockHeight/2$ in the first layer to 0 in centre layer.

For the second file, the z-values of the layers go from 0 to $-totalStockHeight/2$ and the z command becomes:

"G01 Z"+(-((totalStockHeight/2)+slice.layers.get(i).zValue))+""

This is the basic z-command given to the CNC controller, but when using one of the other three roughing strategies, situations arise where the end mill must move up to avoid collisions. When using zig-zag milling for example, the collision array has to be checked for each coordinate. If the position of the path is located in the array, the contour number is sent to a method *getContourHeight()* and the z-value of the layer before this contour first appears is returned. For the second file the z-value of the layer after the contours disappears is returned. This value is then used to write a z-command that moves the end mill to just above the contour, avoiding collision. The same goes for the spiral milling strategies, only these strategies use an array containing path positions where it is safe to move. If a position in the G-code path is not in this array, the end mill is told to move up.

4.6.2 The hole paths

Next are the holes. The list of holes is traversed and for each hole the layers are iterated. Above the middle layer for the first file, and below for

the second, same as before. When a layer is located in the *roughing* list, the current hole's G-code path is traversed. The G-code is written in the same way as for the roughing paths, only this time a different path is used. Once the hole is done and the coordinates have been written for all layers containing that hole, it moves on to the next hole and does the same until all holes have been covered. This way, one hole will be milled at the time, and movement between holes in every layer is avoided.

4.6.3 Rough finishing paths

When writing the G-code for the finishing paths, the same principle applies. Complete one contour before moving on to the next. It basically follows the same formula as when writing G-code for holes, but instead of traversing holes, it traverses contours. Each layer containing that specific contour will then supply a finish path that is written to the file. If a layer is in the list of original roughing layers, i.e. not filler roughing layers, the z-command will be written to zero, making the end mill move up before moving to the next roughing layer. This is to avoid collisions with other contours if e.g. one contour suddenly change shape or/and size.

4.6.4 Final finishing paths

The final finishing paths are written to the G-code files in basically the same fashion as the rough finishing, except now there is a lot more layers, not just roughing layers. The finishing layers are found in the array *finishing*, and each entry in this array contains the contour and which layers needs finishing on this particular contour. The paths themselves are stored in the layer objects.

4.6.5 The middle layer

The last thing written to the G-code files are instructions to mill the middle layer. It is easily retrieved from the list of layers and only consists of a path around one contour.

4.7 Merging programs

To send the G-code to the CNC machine, additional software is required. For this thesis the decision was made to not make this from scratch, and rather use a very capable open source program and merge this with the software implemented in the previous sections.

4.7.1 Universal G-code sender

Universal G-code sender (UGS) is an open source program used to communicate with the CNC controller. Without going into too much detail, the original program can browse for G-code files, and send these to the CNC controller on the CNC machine. In this thesis the program has been altered to accept STL files instead of G-code files. When a STL file has been chosen, the program described in the previous section kicks in. As can be seen in Figure 4.27 the original software has four tabs:

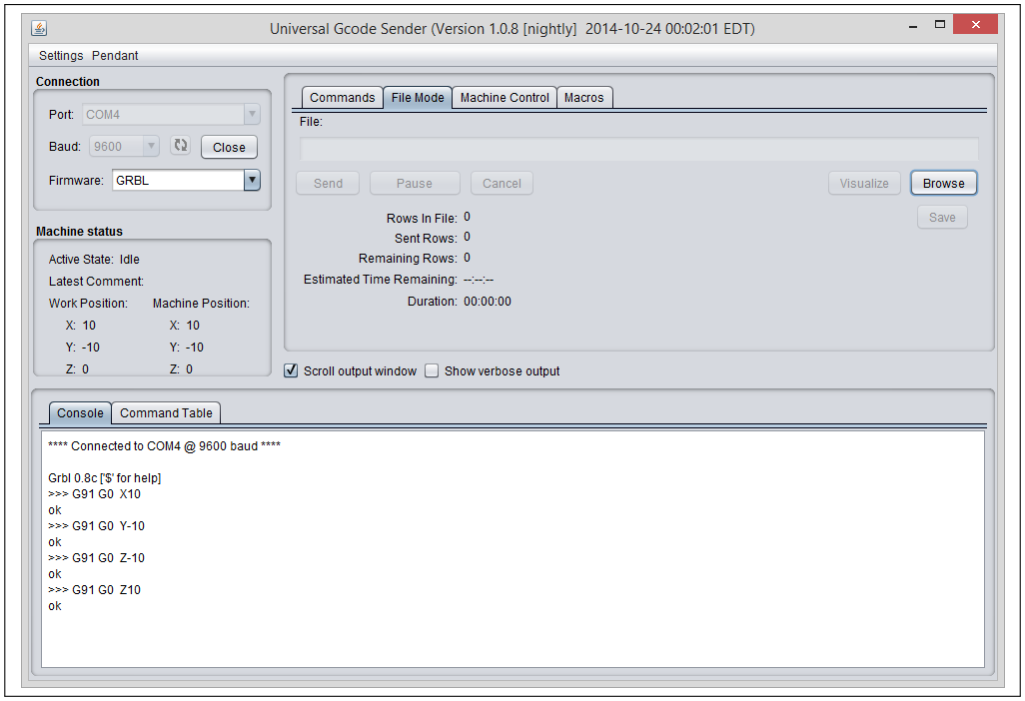

Figure 4.27: Screenshot of the Universal GCode Sender File Mode tab.

- **Commands**: Type in and send a single line of G-code at the time.
- **File Mode**: Browse for an STL file and send it to the CNC controller.
- **Machine Control**: Easy controlling of the CNC machine to get it into starting position. The x,y and z axes can be run with just the push of a button with a chosen millimetre parameter. Figure 4.28 on the next page shows this tab.
- **Macros**: Multiple lines of G-code commands can be written in boxes and run with the pushing a button beside the corresponding box.

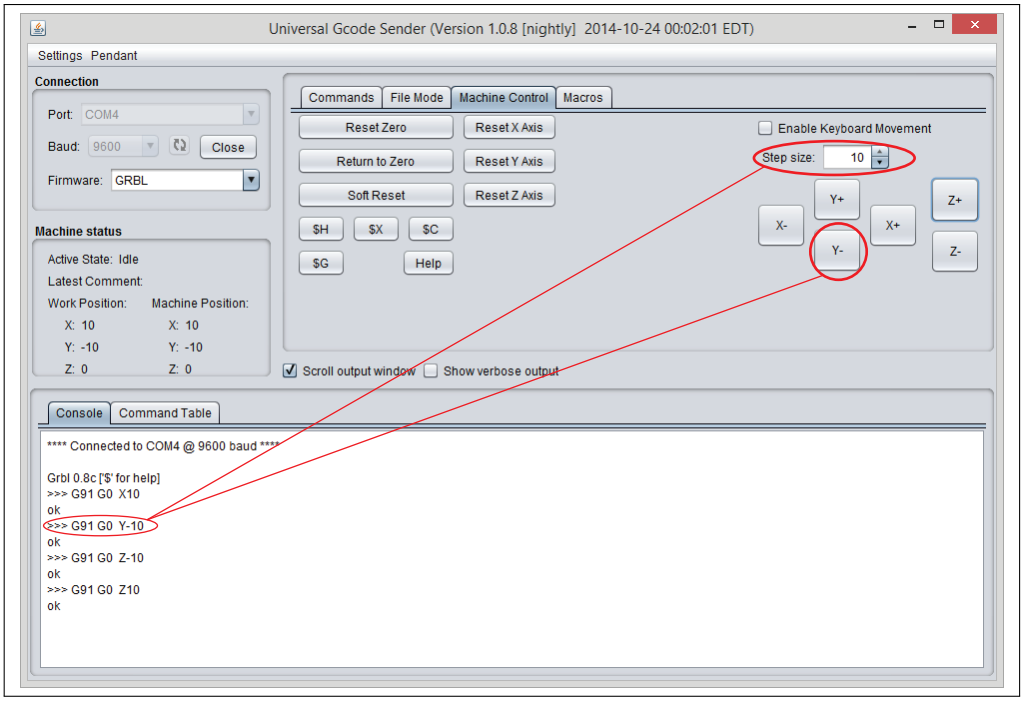

Figure 4.28: Screenshot of the Universal GCode Sender Machine Control tab.

4.7.2 Visualization

Next to the *Browse* button in the file tab is another button called *Visualize*, this can seen in Figure 4.27 on the facing page. When a file has been chosen, this can be pressed and a new window opens that show a 3-dimensional view of the G-code file. Figure 4.29 on the next page shows an example of what this may look like. The yellow line indicates the position of the end mill. White lines are regular G01 commands, while the green ones are movements in the z-direction. When the file is sent to the CNC controller, the yellow indicator will move along the lines corresponding to the movements of the CNC machine, showing the user where the end mill is in the G-code at all times.

4.7.3 BabyMill

For the purpose of this thesis, the UGS software has been altered slightly. As seen in figure 4.30 on page 65 one of the tabs has been removed. The Macros tab seemed unnecessary to keep because it is just not that useful for this thesis and might just end up confusing users. Some radio buttons have been added to allow users to choose a tool diameter and a roughing strategy. Two new buttons *Calculate Path* and *Est. Time* have also been placed beside the *Browse* button. After a STL file has been chosen, the user may press the *Calculate Path* button to calculate and generate the G-

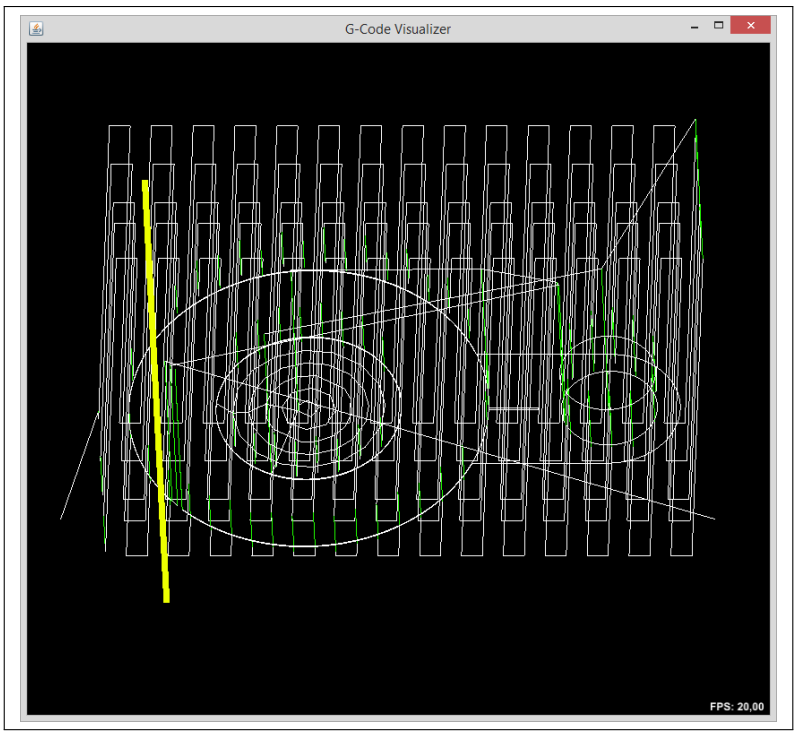

Figure 4.29: Screenshot of the Universal G-Code Sender G-code Visualization feature

code files that can be sent to the CNC controller. Afterwards the *Est. Time* button can be pressed. This starts a little piece of code that goes through the G-code files and estimates the time it will take to mill each side of the model. The time is approximated based on the movement speed of the end mill and distance it has to cover. This is just the theoretical milling time. Due to ignoring the acceleration and deceleration of the milling machine, the time is underestimated[31]. Because of this, a tiny amount of time is added for each line of G-code to compensate. A message with the times will pop up when the button is pressed.

There is already a built in estimated time remaining feature in the original UGS. However, this starts *after* the file has been sent to the CNC controller. It calculates the average time spent on each line of G-code, and creates an estimated remaining time based on how many lines there are left. Since it only makes crude calculations, the estimate will often be off at the end[50]. Since both opening new STL files and calculating paths can take a few seconds, a loading window is presented whenever the user has to wait for something. This is to show that the program is still running and has not crashed, and also to give users something nice to look at while waiting.

Another alteration that has been made can be seen when *Visualization* is pressed after uploading a STL file. The program will generate a G-

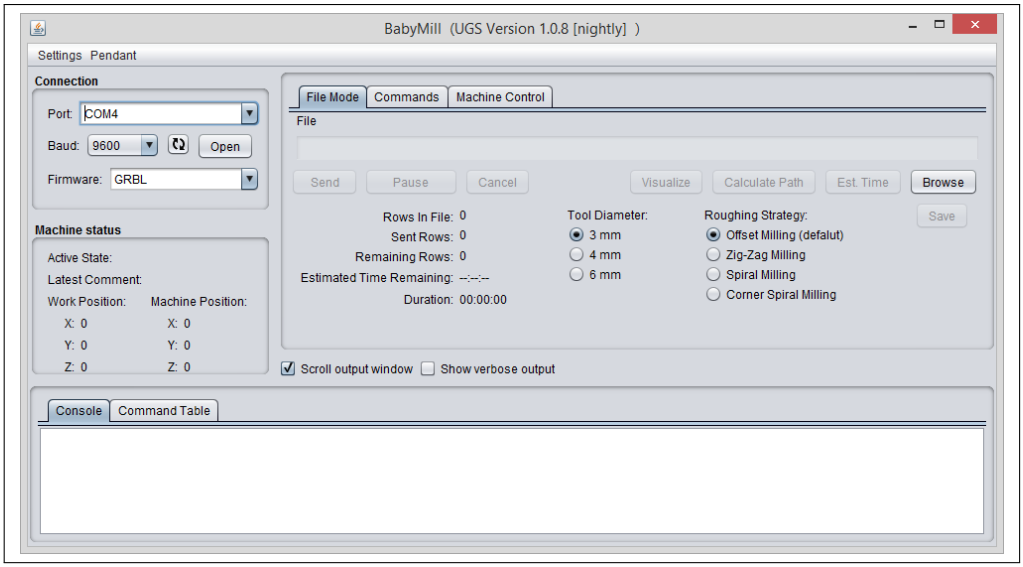

Figure 4.30: Screenshot of the BabyMill File Mode tab.

code file with commands describing all contours and holes in all layers, creating a 3-dimensional image of the model in the STL file. In addition the stock material is also drawn, showing the position of the model in the stock. Figure 4.31 on the next page shows how a model is visualized in the software. Figure 4.32 on the following page shows how g-code is visualized.

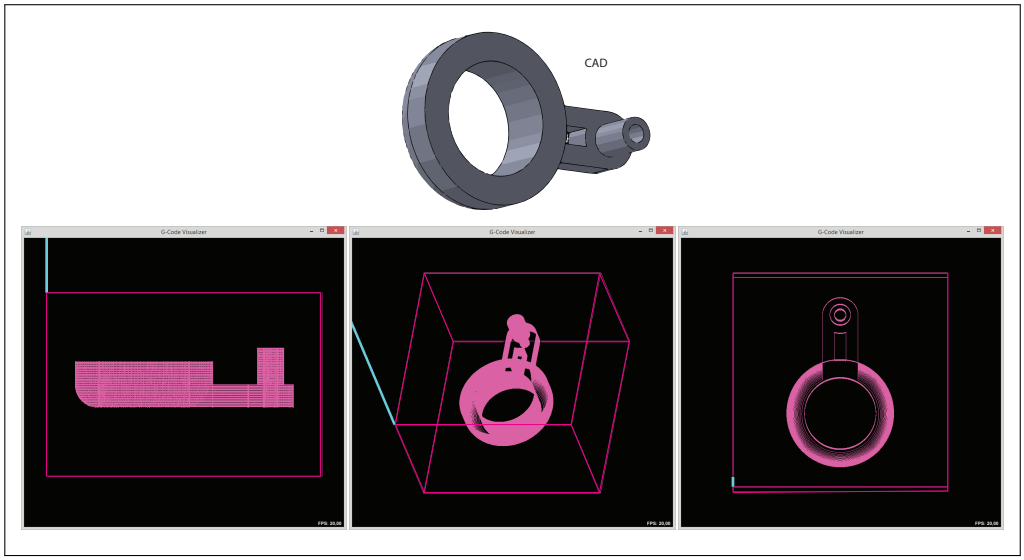

Figure 4.31: Visualization of a model in the BabyMill software. The thick cyan line indicates the starting position of the endmill, i.e. $x = 0$ $y = 0$ $z = 0$.

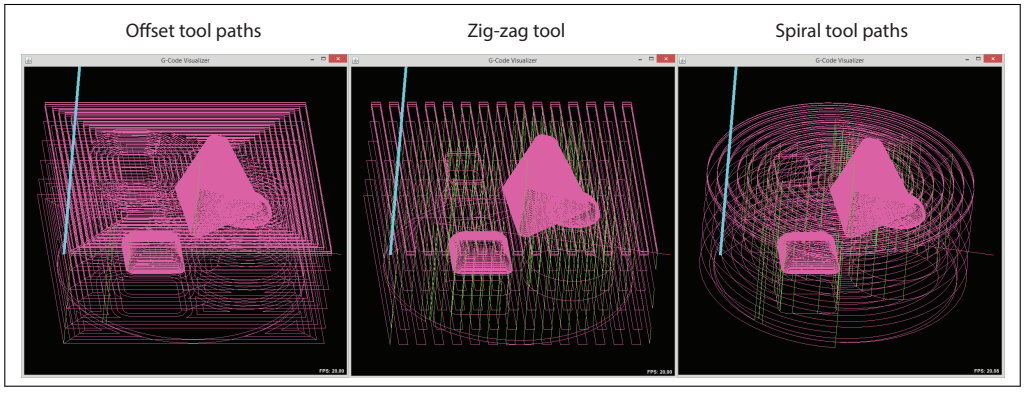

Figure 4.32: Different tool paths generated by the BabyMill software for the same model as used in Figure 2.9 on page 22.

Chapter 5

Testing

This chapter takes a brief look at the practical part of the BabyMill system. It takes a look at the testing set-ups and problems encountered while testing the system on both foam and aluminium.

5.1 Simulator testing

Before testing the g-code produced from the BabyMill software on an actual milling machine, it is tested in a simulator. The g-code simulator used is called CutViewer. It is essential to test the g-code in a simulator first to avoid any damage to milling machines due to faulty code. CutViewer has been a huge asset in discovering when the g-code does something it is not supposed to. This is due to the visualization property it has. When a g-code file has been loaded, a tool diameter and stock size is chosen. A 3D view of the stock and end mill is shown, and the program shows how the end mill interacts with the stock and removes it for every line of code. This makes it is easy to see when something unexpected occurs. Figure 5.1 on the next page gives an idea of how CutViewer visualizes the milling process. The milling simulated is of the first side of a part using the spiral tool path strategy and a 3 mm flat mill.

5.2 Foam testing

After the g-code has been proven machining safe in the simulator, it is tested on the MidiMill with stock material made out foam. Although the parts break easily and cannot be utilized for any practical purposes, foam is a great way to safely test and see what the parts look like when milled.

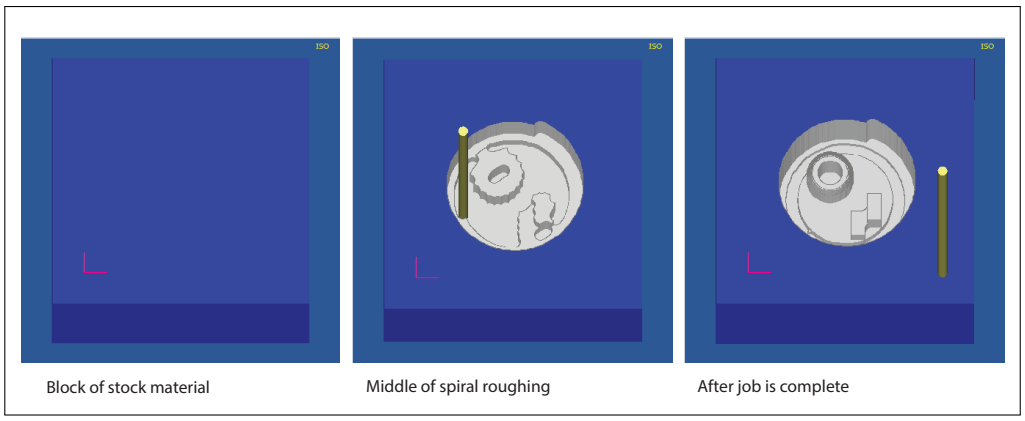

Figure 5.1: Screenshots of the CutViewer 3D simulation.

5.2.1 Stock size

The size of the foam stock for the MidiMill is the largest possible for the machine. The workspace of the MidiMill is approximately $60mm * 60mm * 40mm$, this also becomes the limit for the largest pocket boundary in the BabyMill software. The stock itself has a width of 80 mm, length of 85 mm and height of 40 mm. The reason the stock is not square shaped is to make sure users insert the stock correctly after flipping it. Figure 5.2 shows a stock of foam.

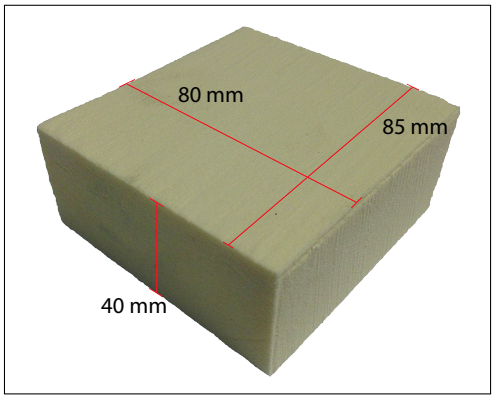

Figure 5.2: Picture of a stock block made of foam.

5.2.2 Work holder

The work table of the MidiMill consists of a grid of small screw holes that work holders can be attached to. Figure 5.3 on the next page shows an image of the work holder for the foam stock.

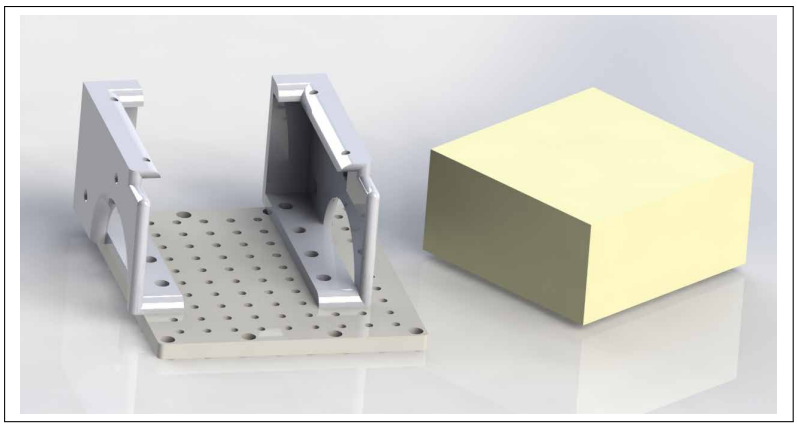

Figure 5.3: Rendered image of the work holder made for the MidiMill

The holes on the top and sides can be used to secure the stock if needed. The foam stock is cut using a hand saw, hence the dimensions can be a little off and some adjustments required. Figure 5.4 on the following page shows the whole milling set-up. The holder itself is printed on a Fortus 250 3D printer.

5.2.3 Problems encountered with foam

When using the MidiMill some problems were encountered:

- **Finding the origin:** Finding the exact origin of the work coordinate system is essential, especially since the stock material is flipped. E.g. 1 mm off on the x-axis will result in a 2 mm overall error. As the stock is flipped about the x-axis, a little inaccuracy on the y-axis is not a problem since it will be the same on both sides. The solution became drawing lines on different parts of the milling machine. When these lines align as seen in Figure 5.4 on the next page, the end mill is in the origin of the coordinate system.
- **Inaccurate stock material:** As a result of the stock being cut by hand, some deviation in x and y directions occur due to stock material not having perfect dimensions. This again causes the part to be a little off as the two sides of the part does not match perfectly. Figure 5.5 on page 71 illustrates the situation.

5.3 Aluminium testing

One of the goals of this thesis is to successfully mill functional parts in aluminium. The aluminium used is called HE30TF Aluminium and is a medium strength alloy with good corrosion resistance[8].

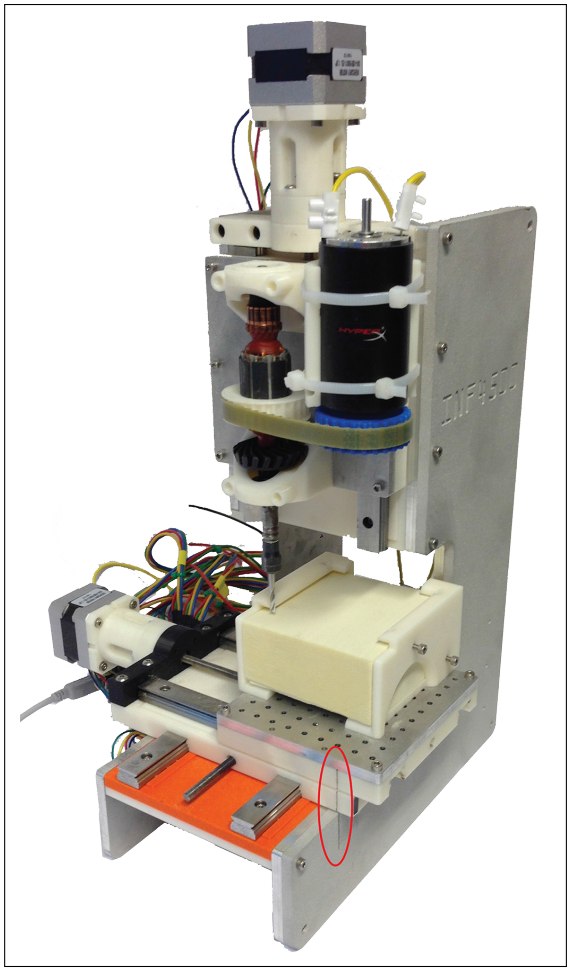

Figure 5.4: Picture of the MidiMill with the foam work holder and a 3 mm end mill in the origin of the work coordinate system.

5.3.1 Stock size

Choosing a stock size for the aluminium was pretty easy since it came in bars with pre-set dimensions. The bars could not be too thick, since the end mill is quite short when milling metals. This is to prevent tool defection[40]. The bar is 1 inch thick, that is 25.4 mm and 2 inches (50.8 mm) wide. It is cut to having a length of 65 mm. Figure 5.6 on the next page shows an aluminium stock.

5.3.2 Work holder

The design of the aluminium work holder is different to that of the foam. This is due to the milling table on the Torjus machine being a t-slot table. The holder this time around is just one solid piece, also printed on the Fortus 250 3D printer. Figure 5.7 on page 72 shows an image of the

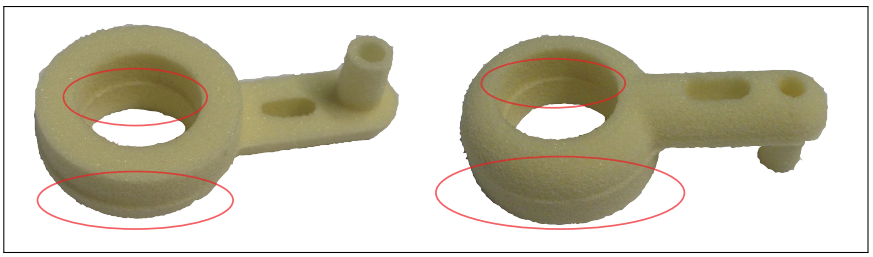

Figure 5.5: Picture of a part where there is a mismatch between the two sides. The mismatch is highlighted by the ellipses

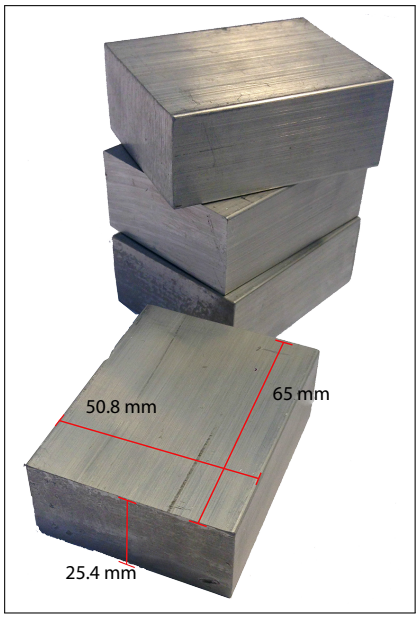

Figure 5.6: Blocks of aluminium stock material.

aluminium work holder. The slots on the sides are used to clamp the holder to the t-slot table. The lines on the top are used to locate the origin of the work coordinate system. Since the aluminium stock blocks have more or less perfect dimensions, they slide right in and no further adjustments are required. However, to keep the stock from sliding out of the work holder while milling, two slots where a pin can be threaded through has been added. The back of the holder is open to make it easier for users to retrieve the stock.

5.3.3 Problems encountered with aluminium

When milling aluminium some new problems were encountered:

- **Heat and chip clearing:** In contrast to foam, aluminium gets quite warm when under stress. When milling without cooling and

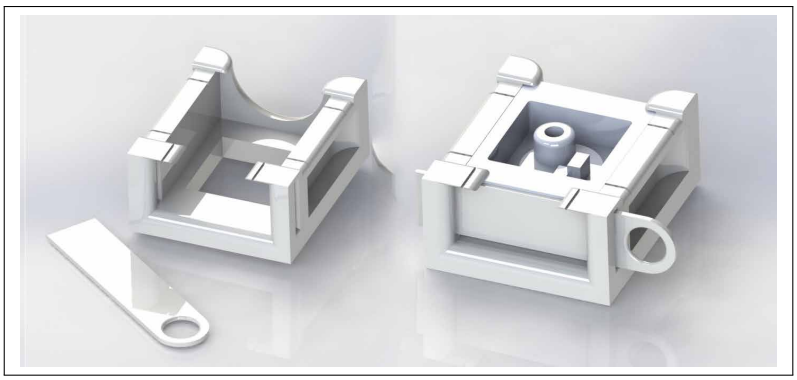

Figure 5.7: Rendered image of the work holder for the aluminium stock.

proper chip clearing, the stock can get so hot it even melts the work holder, moving the stock material inside. This again leads to broken end mills. In addition, when aluminium chips pile up, they can be very hard on the end mills tool life because it has to re-cut already work hardened chips[7]. The solution to these problems became air pressure cooling. It works is by extruding high pressure air at the tip of the end mill. This will both prevent the stock from heating up and will instantly blow away the aluminium chips from the stock material. Figure 5.8 is an example of what happens when milling with and without cooling. Figure 5.9 on the facing page shows the whole set-up for aluminium milling.

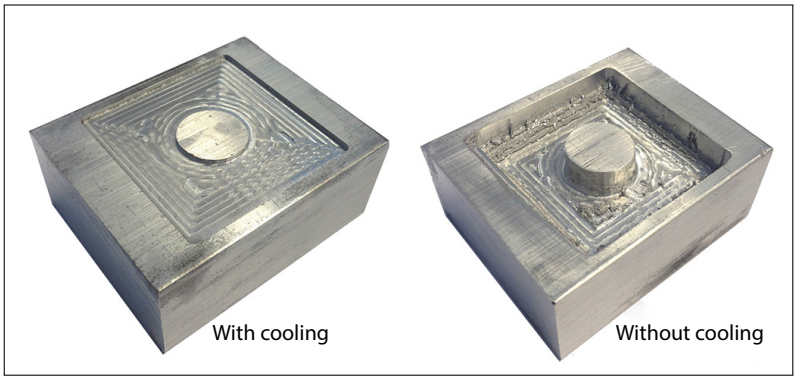

Figure 5.8: Shows the difference when milling with and without cooling.

- **Dust Extraction** A downside with using air pressure as cooling is that it blows the aluminium chips *everywhere*. They cover both the milling machine and the area around it. This can be a problem since the electronic components of the mill are not properly shielded. Worst case there could be a short circuit and a fire while the user is away.

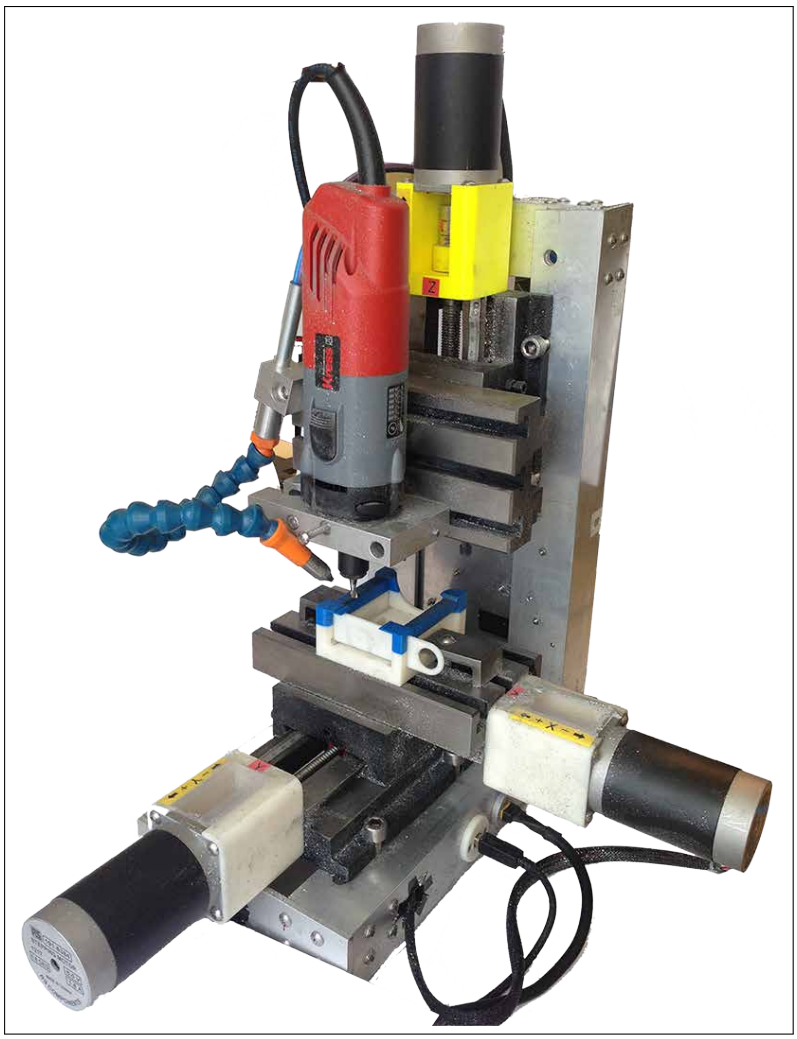

Figure 5.9: Picture of Torjus milling machine with the aluminium work holder and a 6 mm end mill in the origin of the work coordinate system.

Part III

Conclusion

Chapter 6

Results and Analysis

This chapter present and analyses the results found when testing the BabyMill system. First out is simulator accuracy results followed by time results from milling in foam with the different strategies. Then some accuracy and surface finish results from milling aluminium are presented.

6.1 Simulator accuracy results

The CutViewer simulator comes with a measuring tool, allowing users to measure the distance between parallel lines in the model created by the G-code. Figure 6.1 on the next page shows the model measured in SolidWorks and the measuring tool provided by CutViewer. Sadly, the option to measure circles does not work properly, and has therefore not been used. As can be seen in Table 6.1 the G-code created by BabyMill produces dimensions that are spot on in the x-y plane. The difference in Length 2 and Length 2 is probably due to SolidWorks operating with two decimals while CutViewer has three.

	SolidWorks	**Simulator**	**Difference**
Measure			
Length 1	20.99 mm	20.988 mm	-0.002 mm
Length 2	21.44 mm	21.436 mm	- 0.004 mm
Length 3	6 mm	6 mm	0 mm
Length 3	6 mm	6 mm	0 mm

Table 6.1: Simulator accuracy results.

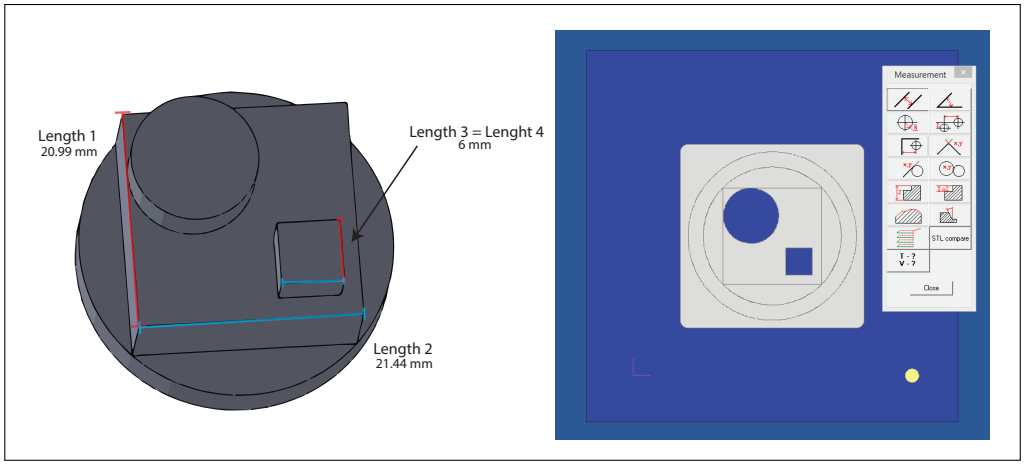

Figure 6.1: Illustrations of a part in SolidWorks and a screenshot of the CutViewer measuring tool.

6.2 Foam time results

Two test parts have been created in SoildWorks and exported to STL. The parts are are made to show some of what can be produced with the BabyMill software.

The corner spiral milling strategy proved to be extremely inefficient because of having to constantly move back and forth between spiral segments. This led to little time spent actually milling away material. Because of this, only the offset, zig-zag and spiral strategies have been used throughout the testing.

6.2.1 First test part

The first test part can be seen in Figure 6.2 on the next page. There is only one side that requires final finishing, the side with the fillet as the middle illustration shows. Figure 6.8 on page 85 shows a picture of the final result.

- **Offset strategy**

  The offset milling strategy does not need the first finishing operations as Tables 6.2 on the next page and 6.3 on page 80 show. The software actually estimated too much time on the first side by about a minute and a half. On the second side it was off by approximately half a minute.

- **Zig-Zag strategy**

  The zig-zag strategy requires both the rough and final finish operations for the first side. For the second only the rough finish is applic-

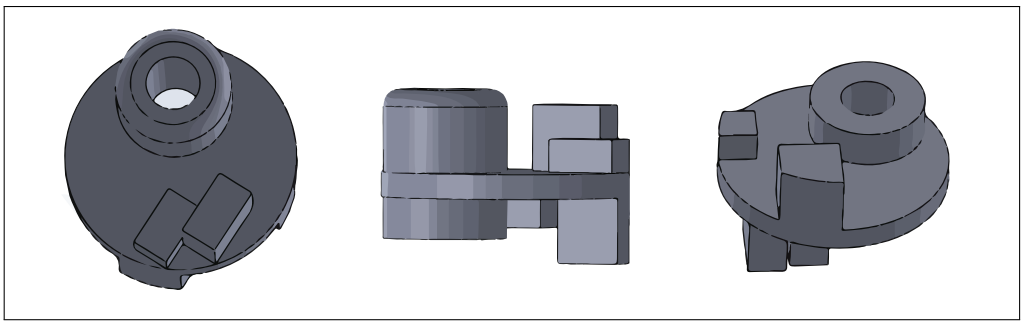

Figure 6.2: Illustrations of the first test part.

	Offset	**Zig-Zag**	**Spiral**
Operation			
First roughing	00:40:56	00:37:57	00:26:52
Rough finish	-	00:04:21	00:04:21
Final finish	00:11:32	00:11:32	00:11:32
Time			
Total time	00:52:28	00:53:50	00:42:45
Estimated time	00:53:00	00:55:00	00:41:00
Difference	- 00:00:32	- 00:01:10	+ 00:01:45

Table 6.2: Time results for the different strategies strategies for the second side of the first test part.

able. The estimated time was very close to the actual time, only 10 seconds off on the first side and half a minute on the second.

- **Spiral strategy**

  This strategy also requires both finishing operations for the first side, and only the rough finish for the second. The estimated time is off by over a minute on both sides as Tables 6.2 and 6.3 show.

- **Comparison**

  If the total times are compared, there is no doubt the spiral strategy is the fastest for this particular part as is easily seen in Figure 6.3. One reason for it being faster than the offset strategy is probably the higher step over. While the offset has a step over of 40% the spiral has approximately a 60% step over. The zig-zag on the other hand has a step over of more than 60%, but still only came in second time wise. This is most likely due to the spiral strategy has a smarter way of avoiding contours. Comparing the times on Table 6.2 the zig-zag even took longer than the offset on side 1 of the part.

	Offset	Zig-Zag	Spiral
Operation			
First roughing	00:59:28	00:39:55	00:34:38
Rough finish	-	00:06:35	00:06:35
Final finish	-	-	-
Time			
Total time	00:59:28	00:46:30	00:41:13
Estimated time	00:57:00	00:48:00	00:40:00
Difference	+ 00:02:28	- 00:01:30	+ 00:01:13

Table 6.3: Time results for the different strategies strategies for the second side of the first test part.

6.2.2 Second test part

Figure 6.4 on page 82 show the second test part. On this part there is need for final finishing operation on both sides due to multiple fillets and the pyramid chamfer. The final result can be seen in Figure 6.8 on page 85.

	Offset	Zig-Zag	Spiral
Operation			
First roughing	00:45:55	00:42:01	00:49:31
Rough finish	-	00:03:37	00:03:37
Final finish	00:31:15	00:31:15	00:31:15
Time			
Total time	01:17:10	01:16:53	01:24:23
Estimated time	01:18:00	01:18:00	01:24:00
Difference	- 00:00:50	- 00:01:07	+ 00:00:37

Table 6.4: Time results for the different strategies strategies for the first side of the second test part.

- **Offset strategy**

  On this test part the offset strategy is apparently the fastest as can be seen in Figure 6.6 on page 84. Tables 6.4 and 6.5 on page 82 show the estimated time is off with about a minute for both sides.

- **Zig-Zag strategy**

  Close behind follows the Zig-zag as the second fastest again for this test part. As Figure 6.6 shows it is the rough finishing operation that puts it behind the offset strategy. Also for the Zig-zag strategy the estimated time is off by plus-minus a minute. Figure 6.5 on page 83 show the result after the different milling operations.

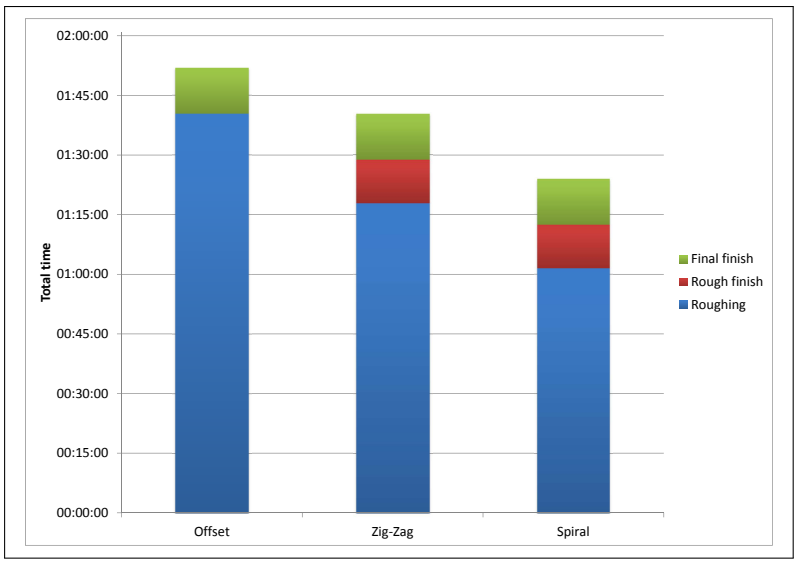

Figure 6.3: A chart displaying the total milling time for the first part using the different strategies.

- **Spiral strategy**

  The spiral strategy proved to be the slowest on this test part as Figure 6.3 shows. However, as Tables 6.4 on the facing page and 6.5 on the next page indicate, its estimated times was the closest to the total times.

- **Comparison**

  The reason spiral strategy is the slowest on the second test part is probably because is has to mill a larger pocket to be able to extract the whole part. This can be seen in Figure 6.7 on page 84 where the sizes of the pockets produced by both the offset and spiral strategies on the two test parts can be compared. This clearly shows the spiral pocket in the top left is considerable larger than that of the same part using offset roughing to the right in the picture.

6.3 Aluminium results

The BabyMill software has also been tested on aluminium. The average step-down distance used when milling aluminium is 1 mm.

6.3.1 Usable part: Accuracy test

The first part milled in aluminium is a part that another student needed for his thesis. As seen in Figure 6.9 on page 85 the part is perfect for the

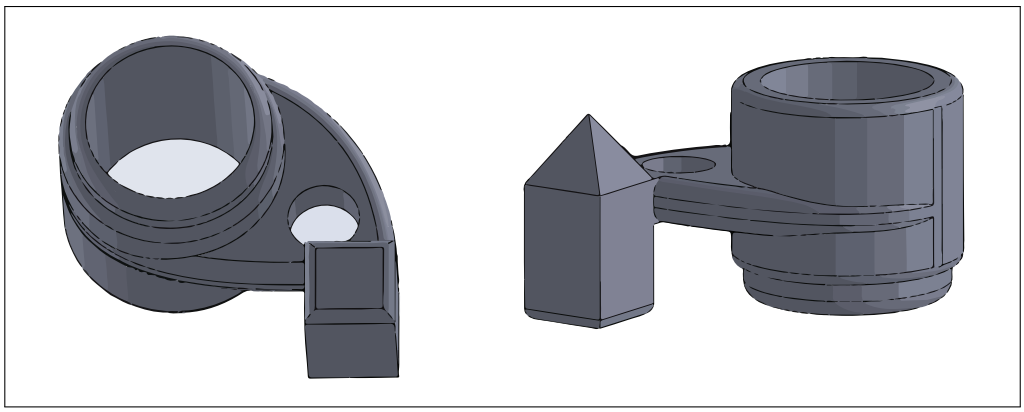

Figure 6.4: Illustrations of the second test part.

	Offset	**Zig-Zag**	**Spiral**
Operation			
First roughing	00:44:55	00:42:51	00:45:16
Rough finish	-	00:03:54	00:03:54
Final finish	00:33:09	00:33:09	00:33:09
Time			
Total time	01:18:04	01:19:54	01:22:19
Estimated time	01:19:00	01:19:00	01:22:00
Difference	- 00:00:56	+ 00:00:54	+ 00:00:19

Table 6.5: Time results for the different strategies strategies for the second side of the second test part.

BabyMill system since it is not too complicated, and it requires to be milled from both sides in order to be realised. After the part was extracted from the stock it was sandblasted to remove the surface markings left behind by the flat end mill seen in Figure 6.10 on page 86. The part was milled with the offset strategy with a 3 mm end mill.

Accuracy analysis

Table 6.6 on the next page shows that the overall accuracy is pretty good. The high difference on height 1 is probably due to the end mill being a little off on the z-axis when the system was set up. Another source of error is that the milling machine has some backlash on the y-axis. Backlash is any kind of unexpected behaviour in an axis because of clearance or looseness of mechanical parts[6]. This caused the round elements of the part to not be perfect circles. The diameters have been measured four different places for each circular element and the listed diameter is the average of these.

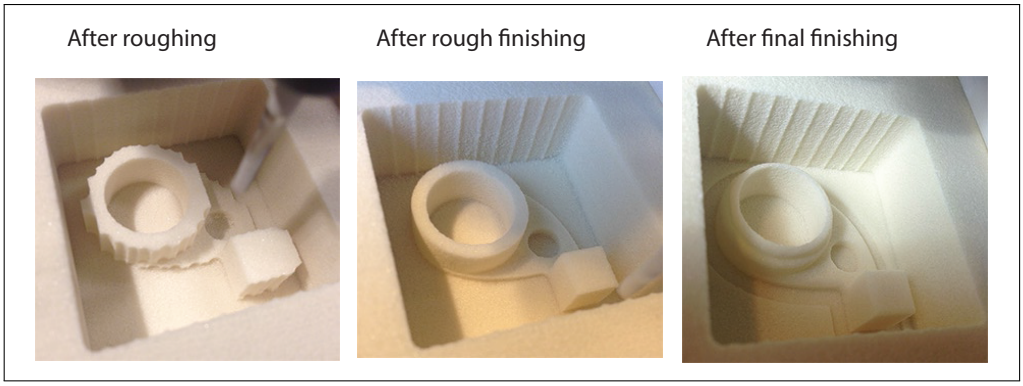

Figure 6.5: Pictures showing the second test part after different milling operations using the Zig-zag roughing strategy.

	Actual	Measured	Difference
Measure			
Height 1	10 mm	9.88 mm	- 0.12 mm
Height 2	3 mm	3.02 mm	+ 0.02 mm
Height 3	3 mm	2.95 mm	- 0.05 mm
Diameter 1	16 mm	16.04 mm	+ 0.04 mm
Diameter 2	6 mm	5.94 mm	- 0.06 mm
Diameter 3	29.80 mm	29.87 mm	+ 0.07 mm
Diameter 4	34 mm	33.97 mm	- 0.03 mm

Table 6.6: Accuracy results for part in aluminium. The measurements can be seen in Figure 6.9 on page 85

6.3.2 Surface finish test

To test the surface finish provided by the final finishing operation, an aluminium sphere was milled. Figure 6.11 on page 86 shows the sphere after different stages in the milling process.

Surface finish analysis

As expected, the middle of the sphere has a quite nice surface finish because of the steep angle as seen in the bottom left picture in Figure 6.11. However, as the angle decreases towards the top of the sphere, the stairs-like scallops can easily be observed. Figure 6.12 on page 87 shows a close-up of the finishing.

Figure 6.13 on page 87 shows a comparison between the second test part 3D printed on the Fortus 250 and milled in aluminium using the BabyMill. The finish on the pyramid section of the part is actually better on the aluminium one.

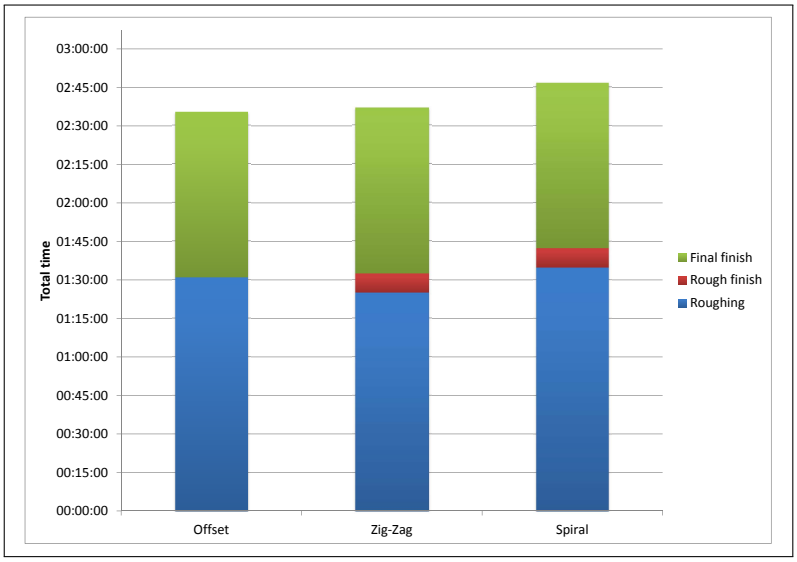

Figure 6.6: A chart displaying the total milling time for the first part using the different strategies.

Figure 6.7: Picture showing the finished parts milled with the offset(right) and spiral(left) strategies.

Figure 6.8: A picture of the finished test parts extracted from the frame of stock material.

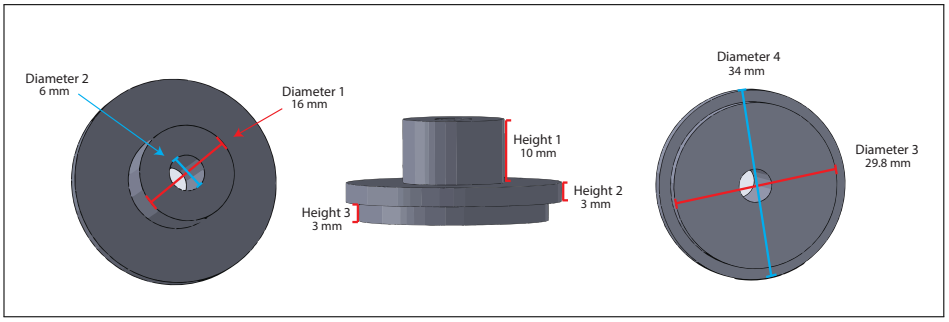

Figure 6.9: An illustration of a usable part with its measurements.

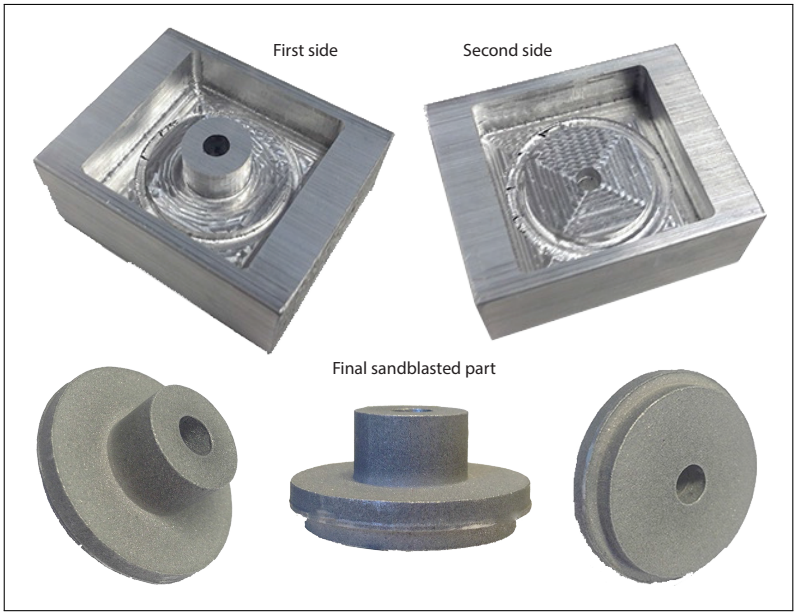

Figure 6.10: A picture of a usable part milled from aluminium.

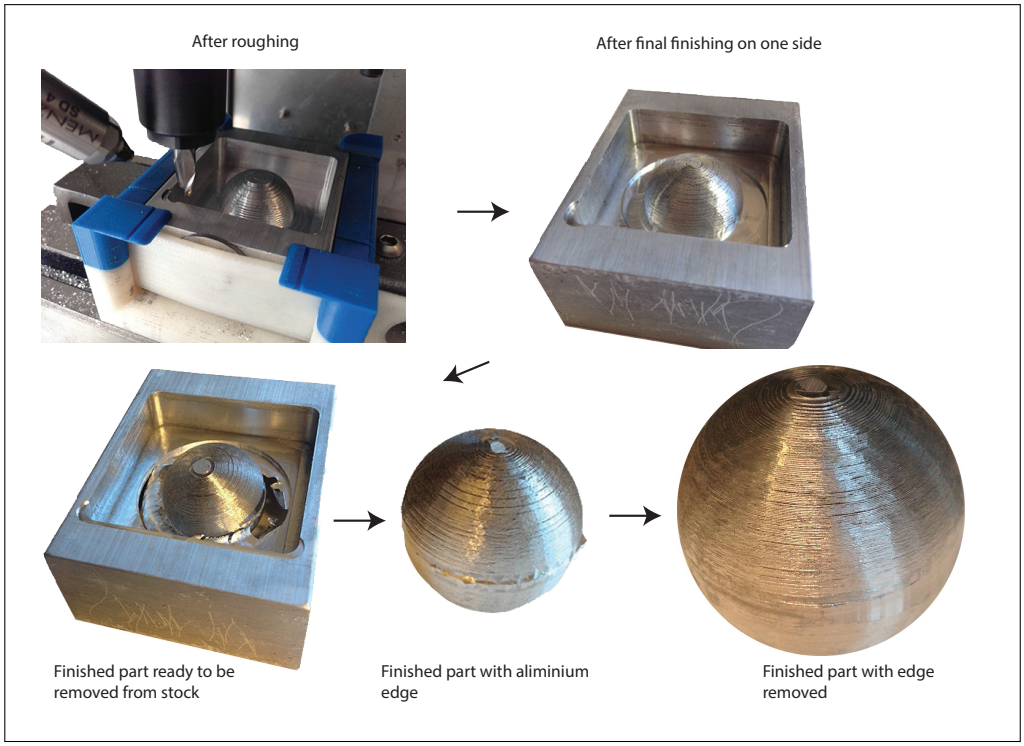

Figure 6.11: Pictures of an aluminium sphere after the different milling operations.

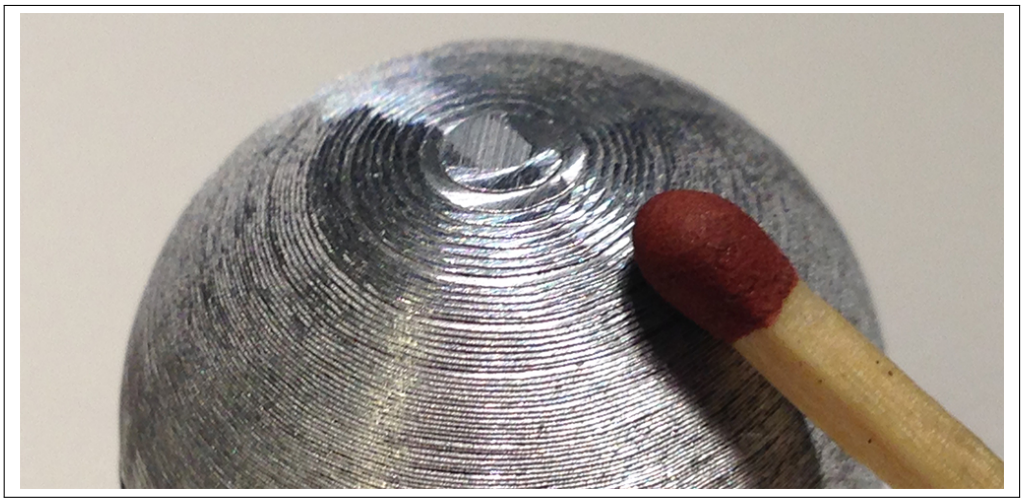

Figure 6.12: Close-up picture of the aluminium sphere. The matchstick is for scale.

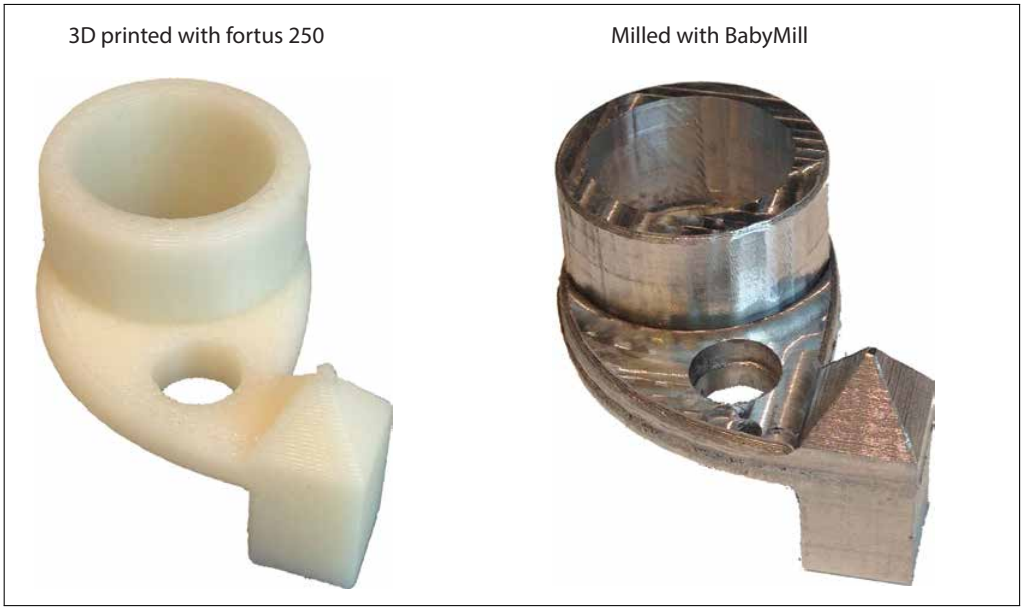

Figure 6.13: The second test part 3D printed and milled in aluminium.

Chapter 7

Discussion and Conclusion

To determine whether or not the BabyMill system delivers on the goals it set out to meet, the criteria for success have to be discussed.

7.1 General Discussion

In this section there is a general discussion around the different aspects of the BabyMill system.

7.1.1 Usability and stability of the software

Since the BabyMill software is an extension of the Universal G-Code Sender, an already user friendly software, it is pretty straight forward to use. When the program starts up the user can immediately browse for a STL file. The part in the file can then be viewed in 3D to verify that the correct part has been selected and that it fits inside the stock material. Next the user can choose between four different roughing strategies and see the estimated time each one of them will take. After a strategy has been chosen, send can be pressed and the milling machine will start processing the part. The user can then observe a visualization of the G-code path with a virtual end mill moving along it synchronously to the real one. All these features really enhance the user experience. Even the loading screen added to indicate that the program is running when files are loaded and tool paths calculated makes it feel less cheap.

The fact that the user gets to choose between strategies is somewhat contradicting the general idea to just browse for a file and push a button. The results from section 6.2 show that the time used by the different roughing strategies varies on the size and shape of the part to be milled. Since each strategy has their strengths and weaknesses, the user has to choose the strategy that takes the shortest time. To avoid confusing users that do not know the differences of the strategies, a further enhancement

of the program would be letting the software itself make the decision on what strategy to use.

Although countless hours have been spent fixing bugs and making sure the majority of different geometries can be handled, this program, like all others in their early stages, is far from perfect. Memory issues may occur when loading the G-code files created, although this happens quite seldom. If a user deliberately tries to crash the program he or she will be able to do so by e.g. loading corrupted files.

7.1.2 File format

The way the STL files are read in the BabyMill software, only information about the different layers of the part are known. Since it is difficult to make assumptions on what the whole part looks like, the choice was made to only implement 2D tool paths for both roughing and finishing. Professional CAM software like HSMWorks uses CAD representations of 3D models and user inputs to determine model geometries and generate 3D tool paths based on that.

For roughing, 2D tool paths are fine, but for finishing, 3D tool paths are preferable because they can provide a better surface finish in a shorter amount of time. However, the results from sections 6.1 and 6.3 shows that parts created with only 2D tool paths are still satisfactory when it comes to surface finish and accuracy.

7.1.3 Strategies

Since the finishing operations are the same for all of the roughing strategies, the final result will be the same regardless of what strategy is used. The main difference between them is the time they use. However, as the milling time depends on the parts, it is difficult to say that one strategy is better than the others. One example is if the height of the part is much smaller than the height of the stock. Then many empty layers have to be cleared of material before reaching the actual part. In this case the zig-zag and spiral strategies will be faster because of the larger step-over. If the part is almost as tall as the stock on the other hand, the tool paths must avoid the contours of the part in all layers. In this scenario the offset strategy will be faster because the other two must move up to avoid collision while the offset tool paths stays in the same layer. Time results found in section 6.2 proves this to be the case, because the second test part is taller than the first one. The results also reveal another factor that affects the time, especially for the spiral strategy. If the part has the shape of a rectangle, the pocket required to extract the part is considerably smaller for the offset and zig-zag strategy. If the part is a perfect circle however, the spiral strategy gets the smallest pocket, but the difference is

not that significant as Figure 7.1 shows.

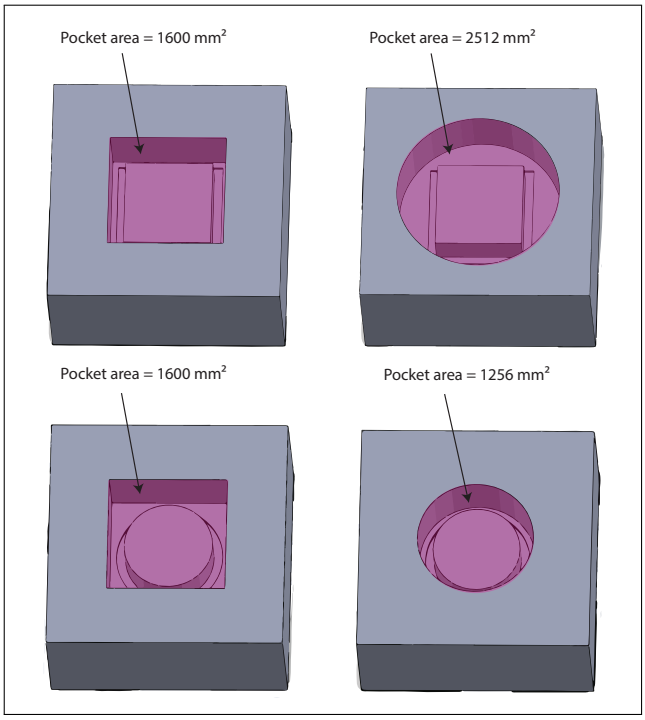

Figure 7.1: Illustration comparing the sizes of milling pockets. The right ones represent spiral strategy while the ones to the left is offset and zig-zag

Another difference between the strategies is how hard they are on the tool. Since there is a lot of plunging and slot milling, the tool is subjected to heavy loads which can cause it to wear faster and maybe even break. However, not enough testing has been done in that area in this thesis. An assumption is that spiral milling is gentler on the tool since it utilizes a large degree of climb milling which is good for prolonged tool life. When using an offset tool path there will be quite a few situations where slot milling takes place. Zig-zag milling will result in conventional milling half the time which can be bad for tool life.

If a strategy could be created by merging two together and combining their strengths, the milling time could be reduced. Two strategies that could go together are the offset and zig-zag. This is because they both create a rectangle shaped pocket. Since the spiral tool paths leave behind a circular pocket, it is difficult combining it with the other two. A good combination could be using zig-zag tool paths on empty layers before reaching the part, and when layers containing contours of the part appear, offset tool paths take over. This way both their strengths are utilized and cancel out each other's weaknesses when it comes to saving time.

7.1.4 Accuracy

From the results acquired in section 6.3 the accuracy is within 0.1 mm which is not that bad. Usually when modelling parts to be printed with an FDM 3D printer approximately 0.1 mm is subtracted from each side due to the plastic expanding and contracting when it is heated or cooled. The results found in section 6.1 go to show that the differences in dimensions is probably caused by the backlash problems of the milling machines a and not by computational errors in the software. However, this is to expect from DIY milling machines. A perfect CNC mill with no backlash whatsoever will be expensive. A solution could be to compensate for the backlash on the milling machine in the software, but since different machines might have different degrees of backlash, this suddenly becomes more difficult to implement.

Another obvious element that affects the accuracy and resemblance the milled part will get to the original is the fact that a round end mill is used. This makes it impossible to get perfect angles without them being rounded as Figure 7.2 show. The rounding of corners could be reduced by decreasing the diameter of the tool used, but doing this would make the milling process take considerably longer time. On this particular problem, the BabyMill must see itself beaten by 3D printers.

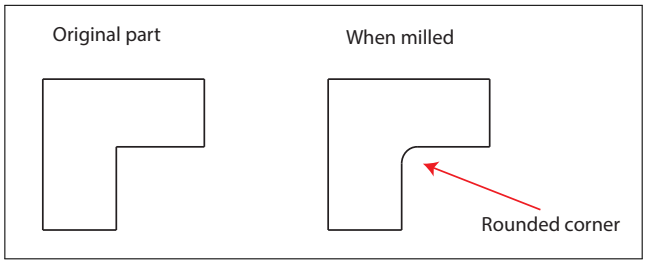

Figure 7.2: Illustrating what inward facing corners looks like after milling

7.1.5 Surface finish

The results from the surface finish test conducted in section 6.3.1 show that the finish is far from perfect on shallow angles. As expected, when using a simple final finishing operation as waterline finishing. If a ball nose end mill had been used during the final finishing process, the finish would have been much better.

Preferably a larger tool is used for roughing to save time, and a smaller for finishing to be able to mill small surface geometries. However, then the user would have to change the end mill, which is not easily done and could quickly ruin the rapid prototyping experience.

7.1.6 Leftover stock

As a result of only using stock material of a certain size, there will be a lot of leftover stock as can be seen in Figure 7.3. Some materials such as metals and plastics could potentially be melted and made into new blocks of stock. Other materials like wood and foam is harder to reuse.

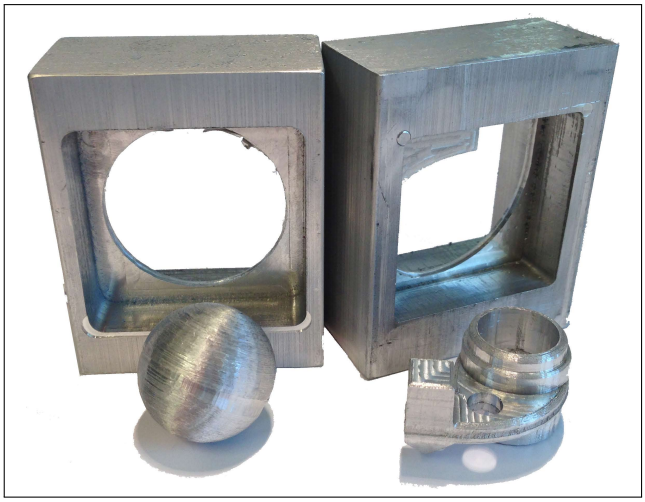

Figure 7.3: Picture showing the leftover stock material.

7.1.7 Mess

An issue that became apparent while testing is that CNC milling is messy. The vacuum cleaner was always nearby. Especially when milling aluminium with air pressure cooling the CNC machine itself and the area around gets covered in small aluminium chips after just one side is milled. This means the user has to dispose of these chips in addition to turning the stock material when one side is done. It takes both time and energy and really hurts the rapid prototyping aspect of the BabyMill system.

A solution could be replacing the air pressure cooling with e.g. a water cooling system. However, when utilizing water cooling, the operation of inserting and turning the stock can suddenly become quite messy. Another option is having a sort of vacuum cleaner system that vacuums the chips spread out from the air pressure before it gets too far.

7.2 Conclusion

The main problem of this thesis questions the possibility to create a rapid prototyping system for CNC milling machines, and see how user friendly it can become by choosing simple strategies and solutions. During this

research project, the BabyMill system has been created. It is a system consisting of a slot solution to easily insert and retrieve stock material, and software that reads STL model files and calculates the necessary tool paths with the click of a button and transmits these directly to the CNC machine.

The BabyMill system has successfully milled parts in both foam and aluminium, with very promising results. To be able to achieve this with just the click of a button, some sacrifices has been made:

- The stock material is of a certain size, if the part is to big, and does not fit in the stock it cannot be milled.
- Only flat end mills can be used. When a size have been chosen, this has to be used for the whole process.
- It utilizes only 2D milling tool paths.

Despite these sacrifices, the BabyMill can produce prototypes of a satisfactory level of quality.

The work holding solution makes for minimal interaction with the CNC machine. In addition the user do not have to think about aligning the stock with a work coordinate system, the stock slides in and everything is set up. The software gives the user a 3D visualization of the models loaded into the program. It also shows a real time simulation of the tool following the tool paths created. Furthermore, it gives a time estimate when the sides are done, so that the user can come back when it is finished and do something else in the meantime.

A goal was to make the BabyMill system as easy to use as a 3D printer. If the CNC machine is already properly set up beforehand, the system is just as user friendly as 3D printing. However, when it comes to convenience, 3D printing is still a few steps ahead the BabyMill:

- When 3D printing, models have no limitations when it comes to shapes, provided support material is used.
- Users do not have to come back halfway through the process, unless the printer runs out of material.
- 3D printers do not require the amount of cleaning up as the BabyMill system does in its current state.

The BabyMill system proves very promising. It can create prototypes in metal with the same accuracy and finish provided by standard FDM 3D printers. With further research and enhancements as the ones discussed in section 7.1, the BabyMill system could very well increase the popularity of CNC machining in the general public.

7.3 Future work

During the work on this thesis, it became apparent that many additions could have been included and should be addressed in further work. As mentioned in the discussion, there should be only one milling strategy. The program decides which one to use, with the possibility to merge milling strategies together to save time.

As the software is now, the step-down for the final finishing operation are set to 0.1 mm. The user should be able to override this, by choosing a poorer or better finish depending on the time the user has at his or her disposal. Another option could be to add adaptive step-down. In areas with steep angles the step-down should be higher than in areas with shallow angels.

When it comes to physical improvements, a way to extract the dust and chips produced by the milling machine would make the system more convenient to use. Another thing to further increase usability is a device that flips the work-holder shelf when one side is done so the user will not have to. However, by adding these, the system will not be easy to set up with a new milling machine.

More research is needed on how the different roughing strategies affect the tool used. This way it is possible to determine if higher step-down between roughing layers can be used without risking breaking the tool. Fewer roughing layers would result in reduced milling time.

Appendix A

First appendix

The complete source code and the program itself can be found at:
http://www.robotikk.com/student/projects/Thomas%20Benjaminsen_RP_CNC/

Source code for the reading of STL files and sorting of the boundary lines.

```
package com.thomkbe.slice;
import java.applet.Applet;
import java.io.File;
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.InputStream;
import java.io.DataInputStream;
import java.io.FilterInputStream;
import java.applet.*;
import java.awt.*;
import java.lang.Math.*;
import java.util.HashMap;
import java.util.Collections;
import java.awt.geom.AffineTransform;
import java.awt.BasicStroke;
import java.awt.Event;
import com.vividsolutions.jts.geom.Coordinate;
import com.vividsolutions.jts.geom.GeometryFactory;
import com.vividsolutions.jts.geom.LinearRing;
import com.vividsolutions.jts.geom.Point;
import com.vividsolutions.jts.geom.Polygon;
import com.vividsolutions.jts.geom.MultiPolygon;
import com.vividsolutions.jts.geom.LineString;
import com.vividsolutions.jts.geom.Envelope;
import com.vividsolutions.jts.geom.LineSegment;
```

```
import com.vividsolutions.jts.operation.buffer.*;
import com.vividsolutions.jts.geom.PrecisionModel;
import com.vividsolutions.jts.geom.GeometryCollection;
import com.vividsolutions.jts.geom.Geometry;
import com.willwinder.universalgcodesender.MainWindow;
/**
 *
 *@author Thomas
 */

class Slicer{

 //Initiate variables
 float z_min = 9999;
 float z_max = -9999;
 double totalStockHeight;
 double z_c;
 float f1x,f1y,f1z;
 int v1;
 DataInputStream in;
 byte[] fourbytes;
 int antTriangles;
 int antLayers;
 boolean endOfFile;
 boolean skip;
 boolean sort;
 boolean set;
 boolean findNewContour;
 boolean second;
 float[][] v = new float[3][3];
 float[] facet = new float[3];
 int intersectionPointCnt = 0;
 int lastIntersectionPointCnt = 0;
 HashMap map = new HashMap();
 HashMap map2 = new HashMap();
 ArrayList<String> list = new ArrayList<String>(1000);
 boolean endit = false;
 int antSkip = 0;
 int whereisz_min = 0;
 float middle;
 String lastKey, lastKey2;
 String key;
 int whatMap = 0;
 boolean skipLayer;
 Coordinate[] tmpCordPath;
 int sortedInContour = 0;
 MapVector mv;
```

```
int lastContCnt = 0;

float totalHeight = 0;
ArrayList newContourLayers = new ArrayList(1000);
int antEmpty = 0;
float tmp = 0;
float tmp2 = 0;
Vector3 p0tmp,p1tmp;
int tmpcnt = 0;
int antSorted = 0;
int antSorted2 = 0;

//Initiate the HashMap and ArrayLists
HashMap<String,Path> layerPath = new HashMap(1000);

ArrayList<Triangle> triangles;
ArrayList<Layer> layers;
ArrayList<Triangle> triangleList=new
   ArrayList<Triangle>(1000);


//Create a Slicer object
public Slicer(File file, float middle, double
   totalStockHeight){
 this.middle = middle;
 this. totalStockHeight = totalStockHeight;
 fourbytes = new byte[4];
 System.out.println("Saving triangles");

 try {
  in = new DataInputStream(new FileInputStream(file));
 } catch (FileNotFoundException e) {
  System.err.println("An error occured when trying to open
     the file "+filename);
 }

 try {
  in.skipBytes(80);
 } catch (IOException e1) {
  // TODO Auto-generated catch block
  e1.printStackTrace();
 }

 try {
  in.read(fourbytes);
 } catch (IOException e1) {
  // TODO Auto-generated catch block
```

```
 e1.printStackTrace();
}
//read number of triangles
//Is little endian so need to rearrange the order of the
   bytes.

antTriangles=((fourbytes[3]&0xff)<<24) +
   ((fourbytes[2]&0xff)<<16)+ ((fourbytes[1]&0xff)<<8) +
   (fourbytes[0]&0xff);
triangles = new ArrayList<Triangle>(antTriangles);

System.out.println("antTriangles = "+antTriangles);

int vNr = -1;
//traverse all the triangles in the file
for(int i = 0;i < antTriangles * 4;i++){

 if(i%4 == 0&& i != 0){
  skip=true;
  try {
   in.skipBytes(2);
  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
 }

 try {
  in.readFully(fourbytes);
 } catch (IOException e) {
  // TODO Auto-generated catch block
 }

 v1 = ((fourbytes[3]&0xff) << 24) | ((fourbytes[2]&0xff)
    << 16) |
      ((fourbytes[1]&0xff) << 8) | ((fourbytes[0]&0xff)
         << 0);
 f1x = Float.intBitsToFloat(v1);
 if(i%4==0)facet[0]=f1x;
 else v[vNr][0]=f1x;
 try {
  in.readFully(fourbytes);
 } catch (IOException e) {
  // TODO Auto-generated catch block
  e.printStackTrace();
 }
 v1 = ((fourbytes[3]&0xff) << 24) | ((fourbytes[2]&0xff)
```

```
      << 16) |
        ((fourbytes[1]&0xff) << 8) | ((fourbytes[0]&0xff)
           << 0);
  f1y = Float.intBitsToFloat(v1);
  if(i%4==0)facet[1]=f1y;
  else v[vNr][1]=f1y;
  try {
   in.readFully(fourbytes);
  } catch (IOException e) {
   // TODO Auto-generated catch block
   e.printStackTrace();
  }
  v1 = ((fourbytes[3]&0xff) << 24) | ((fourbytes[2]&0xff)
      << 16) |
        ((fourbytes[1]&0xff) << 8) | ((fourbytes[0]&0xff)
           << 0);
  f1z = Float.intBitsToFloat(v1);
  if(i%4==0) facet[2] = f1z;
  else v[vNr][2] = f1z;
  if(++vNr == 3){
   vNr = -1;
   triangles.add(new Triangle(v,facet));
  }

  if(!skip){
   if(f1z < z_min)z_min = f1z;
   if(f1z > z_max)z_max = f1z;
  }
  skip=false;
 }

 System.out.println("z_min= "+z_min+" z_max= "+z_max);
 totalHeight = z_max-z_min;
}
//method to print the triangles with vertex info
public void printTriangles(){
 System.out.println("---------------------------------");
 for(int i = 0;i < antTriangles;i++){
  System.out.println("Triangle "+(i+1));
  System.out.print("Facet: ");
  for(int j = 0;j < 3;j++){
   System.out.print(triangles.get(i).facet[j]+" ");
  }
  System.out.println();
  for(int j = 0;j < 3;j++){
   System.out.print("Vertex"+(j+1)+": ");
   for(int k = 0;k < 3;k++){
```

```
   System.out.print(triangles.get(i).v[j][k]+" ");
  }
  if(triangles.get(i).v[j][2] == z_min)whereisz_min = i;
  System.out.println();
 }
 System.out.println();
}
}

// traverses the triangle list to find the minimum z-value
public void findZ_min(){
 double tmpZ_min = 99999;
 for(int i = 0;i < antTriangles;i++){
  for(int j = 0;j < 3;j++){
   if(triangles.get(i).v[j][2] < tmpZ_min) tmpZ_min =
      triangles.get(i).v[j][2];
  }
 }
 z_min = (float)tmpZ_min;
}

// changes the orientation of the model
// by swapping x and z-values in all triangles
public void changeOrientation(){
 float tmp_value;
 z_min = 9999999;
 z_max = 0;

 for(int i = 0;i < triangles.size();i++){
  for(int j = 0;j < 3;j++){
   tmp_value = -triangles.get(i).v[j][2];
   triangles.get(i).v[j][2] = triangles.get(i).v[j][0];
   triangles.get(i).v[j][0] = tmp_value;
   if(triangles.get(i).v[j][2] < z_min) z_min =
      triangles.get(i).v[j][2];
   if(triangles.get(i).v[j][2] > z_max) z_max =
      triangles.get(i).v[j][2];
  }
 }
 System.out.println("NEW afer orientation change: z_min=
    "+z_min+" z_max= "+z_max);
 totalHeight = z_max - z_min;
}

// Creates the layers
public void makeLayers(float layerThickness){
 float tmpZ = (float)(totalStockHeight / 2);
```

```
double maxmax = -999;
double minmin = 999;
int teller = 1;
int contCnt = 0;
int contourStart = 0;
antLayers = 0;

// if z_min !=0, need to adjust the part so that bottom
   is at 0.
System.out.println("Z_min: "+z_min+" Z_max: "+z_max);
if(middle != 0){
 System.out.println("Needs adjustment!");
 for(int i = 0;i < antTriangles;i++){
  for(int j = 0;j < 3;j++){
   triangles.get(i).v[j][2] = triangles.get(i).v[j][2] -
       (z_max/2) - (tmpZ + middle);
   if(triangles.get(i).v[j][2] > maxmax) maxmax =
       triangles.get(i).v[j][2];
   if(triangles.get(i).v[j][2] < minmin) minmin =
       triangles.get(i).v[j][2];
  }
 }

 if(maxmax > (totalStockHeight / 2) || minmin <
    -(totalStockHeight / 2)){
  System.out.println();
  if(maxmax > (totalStockHeight / 2))
     System.out.println("maxmax = "+maxmax+ "
     "+((float)maxmax-((float)totalStockHeight / 2)));
  if(minmin<-(totalStockHeight /
     2))System.out.println("minmin = "+minmin+"
     "+-(((float)totalStockHeight / 2) + (float)minmin));
  System.out.println();
  for(int i = 0;i < antTriangles;i++){
   for(int j = 0;j < 3;j++){
    if(maxmax > (totalStockHeight /
       2))triangles.get(i).v[j][2] =
       triangles.get(i).v[j][2] - ((float)maxmax -
       ((float)totalStockHeight / 2) + (float)0.5);
    if(minmin < -(totalStockHeight /
       2))triangles.get(i).v[j][2] =
       triangles.get(i).v[j][2] -
       (((float)totalStockHeight / 2) + (float)minmin) +
       (float)0.5;
   }
  }
```

```
 }
 z_min = 0;
 System.out.println("middle: "+middle);
 System.out.println("tmpZ: "+tmpZ);
 System.out.println("tmpZ - middle: "+(tmpZ-middle));

}else{
 for(int i = 0;i < antTriangles;i++){
  for(int j = 0;j < 3;j++){
   triangles.get(i).v[j][2] = triangles.get(i).v[j][2] -
       (z_max / 2);
  }
 }
 z_min = 0;
}

while(tmpZ >= (-(totalStockHeight / 2))){
 antLayers++;
 tmpZ -= layerThickness;
}
tmpZ = (float)(totalStockHeight/2);
layers = new ArrayList<Layer>(antLayers);
System.out.println("antLayers= "+antLayers);
for(int i = 0;i < antLayers;i++){
 layers.add(new Layer(tmpZ));
 tmpZ -= layerThickness;
}

tmpZ = (float)(totalStockHeight/2);

// slice the triangles using the z-values from the
// layers as planes.

for(int i = 0;i < antLayers;i++){
 Vector3 vPlaneDirection = new Vector3( 0, 0,
    layers.get(i).zValue );
 Vector3 vPointOnPlane = new Vector3( 0, 0,
    layers.get(i).zValue);
 intersectionPointCnt = 0;
 triangleList.clear();
 for(int j = 0;j < antTriangles;j++){
  int a = clipTriangle(triangles.get(j), vPlaneDirection,
     vPointOnPlane,i);
 }
 System.out.println("layer: "+i+" intersectionPointCnt=
    "+intersectionPointCnt);
 tmpCordPath = new Coordinate[intersectionPointCnt+1];
```

```
// all intersection points are now located and
// stored in the HashMap
contCnt = 0;

// only enter if there are contours in the layer
if(intersectionPointCnt > 0 && !layers.get(i).skipLayer){
 layers.get(i).contours[contCnt] = new Path();
 // retrieve a random key
 key = list.get(0);
 if(map.containsKey(key)){
  mv = (MapVector)map.get(key);
  whatMap = 1;
 }
 tmpCordPath[sortedInContour++] = new
    Coordinate(mv.one.m_x,mv.one.m_y);

 // while there are still intersection points that are
 // not sorted
 while(antSorted < intersectionPointCnt && map.size() >
    0){
  antSorted++;
  // sort the first two points as a new coordinate
  tmpCordPath[sortedInContour++] = new
     Coordinate(mv.two.m_x,mv.two.m_y);
  MapVector tmp = (MapVector)map.get(mv.keyToOther);
  // remove key from HashMap and list
  map.remove(key);
  map.remove(mv.keyToOther);
  list.remove(key);
  list.remove(mv.keyToOther);

  key = ""+(double)mv.two.m_x * 10000+"
     "+(double)mv.two.m_y * 10000+"";

  if(!map.containsKey(key)){
   lastKey = key;
   key += " 1";
  }
  // retrieve next MapVector
  if(map.containsKey(key)){
   mv = (MapVector)map.get(key);
   whatMap = 1;

  // if key is not located in the HashMap, but still
  // not finished
  }else if(antSorted<intersectionPointCnt&&map.size()>0){
```

```
  layers.get(i).contours[contCnt].thePathCords = new
     Coordinate[sortedInContour];
  System.arraycopy(tmpCordPath, 0,
     layers.get(i).contours[contCnt].thePathCords,
     0,sortedInContour-1);
  // Save contour found
  layers.get(i).contours[contCnt].
     thePathCords[sortedInContour - 1] = new
     Coordinate(layers.get(i).contours[contCnt].
     thePathCords[0].x,
     layers.get(i).contours[contCnt].thePathCords[0].y);
  tmpCordPath = new Coordinate[intersectionPointCnt +
     1];
  sortedInContour = 0;
  // start finding a new contour
  newCont(i);
  if(endit)break;
  if(map.containsKey(key) || map2.containsKey(key)){
   contCnt++;
   layers.get(i).contours[contCnt] = new Path();
  }
  tmpCordPath[sortedInContour++] = new
     Coordinate(mv.one.m_x,mv.one.m_y);
 }
}
endit = false;

layers.get(i).contours[contCnt]. thePathCords = new
   Coordinate[sortedInContour];
System.arraycopy(tmpCordPath, 0,
   layers.get(i).contours[contCnt].thePathCords,
   0,sortedInContour-1);

layers.get(i). contours[contCnt].
   thePathCords[sortedInContour-1] = new
   Coordinate(layers.get(i). contours[contCnt].
   thePathCords[0].x, layers.get(i).contours[contCnt].
   thePathCords[0].y);

tmpCordPath = new Coordinate[intersectionPointCnt + 1];
sortedInContour = 0;
}

// when done with a layer save contour in the layer class
list.clear();
map.clear();
map2.clear();
```

```
System.out.println("Ant sorted: "+antSorted);
antSorted=0;
System.out.println("antSorted:
   "+layers.get(i).antSorted+ " contCnt= "+contCnt);
layers.get(i).contourCnt = contCnt;
intersectionPointCnt = 0;
boolean done = false;
int tmpj = 0;
if(layers.get(i).skipLayer&&i != 0){
 Layer tmpLayer = new Layer(tmpZ);
 for(int j = 1;j < 10;j++){
  if(i-j > 0 && !layers.get(i-j).skipLayer){
   if(layers.get(i-j).contourCnt != 0){
    for(int k = 0;k <= layers.get(i-j).contourCnt;k++){
     System.out.println("k: "+k);
     tmpLayer.contours[k] = new Path();
     tmpLayer.contours[k].thePathCords = new
        Coordinate[layers.get(i -
        j).contours[k].thePathCords.length];
     for(int l = 0;l < layers.get(i -
        j).contours[k].thePathCords.length ; l++){
      tmpLayer.contours[k].thePathCords[l] = new
         Coordinate(layers.get(i -
         j).contours[k].thePathCords[l].x , layers.get(i
         - j).contours[k].thePathCords[l].y);
     }
    }
   }
   tmpj = j;
   tmpLayer.contourCnt = layers.get(i - j).contourCnt;
   System.out.println("skip layer: "+i+" Replace with
      layer: "+(i-j));

   antSkip++;
   done = true;
   break;
  }
  if(done)break;
 }
 layers.set(i, tmpLayer);
 System.out.println("layers.get("+ i +").contourCnt:
    "+layers.get(i).contourCnt);
 System.out.println("layers.get("+(i -
    tmpj)+").contourCnt:
    "+layers.get(i-tmpj).contourCnt);
}
tmpZ -= layerThickness;
```

```
 }
}
// starts finding a new contour by getting a
// random key from the list of keys
public void newCont(int layer){
 boolean done = false;
 key = list.get(0);
 if(map.containsKey(key)){
  mv = (MapVector)map.get(key);
  done = true;
  whatMap = 1;
 }else{
  endit = true;
  System.out.println("Key not in map!");
 }
}


// Original author of the triangle clipping algorithm is
   Ben Kenwright

// it takes a triangle, and a plane, and slices the
// triange.. two intersection points Returns 0 if it
   didn't cut
// the triangle - 1 or 2 depending on how many times it
   cut the triangle
// on the positive side of the triangle.
// Returns 3 for all on the positve side of plane
// Returns 0 for all on the negative side of plane

public float distRayPlane(Vector3 vStart, Vector3 vEnd,
   Vector3 vnPlaneNormal, Vector3 vPointOnPlane){
 float cosAlpha;
 float deltaD;
 float planeD = Vector3.dot( vnPlaneNormal, vPointOnPlane
    );

 Vector3 vRayVector = Vector3.subtract(vEnd, vStart);
 Vector3 vnRayVector = Vector3.normalize(vRayVector);
 cosAlpha = Vector3.dot( vnPlaneNormal, vnRayVector );
 // parallel to the plane (alpha=90)
 if ( Math.abs(cosAlpha) < 0.001f ) return
    Vector3.length(vRayVector);

 deltaD = planeD - Vector3.dot(vStart, vnPlaneNormal);

 float l = (deltaD/cosAlpha);
```

```
 return l;
}


public Vector3 PointOnPlane( Vector3 vStart, Vector3
   vEnd, Vector3 vPlaneNormal, Vector3 vPointOnPlane ){
  Vector3 vn = Vector3.subtract(vEnd,vStart);
  vn = Vector3.normalize(vn);
  float Length =
     distRayPlane(vStart,vEnd,vPlaneNormal,vPointOnPlane);
  Vector3 px = Vector3.scale( vn, Length );
  px = Vector3.add( vStart, px );

  return px;
}

int clipTriangle( Triangle t, Vector3 vPointOnPlane,
   Vector3 vPlaneNormal, int layer){
  // Checking - not necessary though
  vPlaneNormal = Vector3.normalize(vPlaneNormal);

  // Clipping
  Vector3 p0 = new Vector3( t.v[0][0], t.v[0][1],
     t.v[0][2]);
  Vector3 p1 = new Vector3( t.v[1][0], t.v[1][1],
     t.v[1][2]);
  Vector3 p2 = new Vector3( t.v[2][0], t.v[2][1],
     t.v[2][2]);

  Vector3 v1 = Vector3.subtract( p1, p0 );
  Vector3 v2 = Vector3.subtract( p2, p0 );

  //see if the triangle actually cuts the plane
  float k = Vector3.dot( vPlaneNormal, vPointOnPlane );

  float a0 = Vector3.dot( vPlaneNormal, p0 );
  float a1 = Vector3.dot( vPlaneNormal, p1 );
  float a2 = Vector3.dot( vPlaneNormal, p2 );

  // Determine how many points are on the positive side of
     our plane
  int iCount = 0;
  boolean p0in = false;
  boolean p1in = false;
  boolean p2in = false;
  if( (k - a0) > 0 ){ iCount++; p0in = true; }
  if( (k - a1) > 0 ){ iCount++; p1in = true; }
```

```
if( (k - a2) > 0 ){ iCount++; p2in = true; }

Vector3 px0 = new Vector3();
Vector3 px1 = new Vector3();

// If the triangle is fully on one side, or fully on the
   other side
if( iCount==0) return 3; // All on the positive side on
   plane
 if( iCount==3) return 0; // all on the negative side of
    plane


// These two vectors will hold the two points on the
   plane that
// actually cut the triangle.
// For example if we go from PointA to PointB on our
   triangle,
// and it goes through the plane, the point where A->B
   cuts the
// plane is called px0 for example.


// Now we have the two intersection points on the plane
// slicing the triangle

if(iCount==1 )
{
 Vector3 pA = new Vector3();
 Vector3 pB = new Vector3();
 Vector3 pC = new Vector3();

 if( p0in ){ pA = p0; pB = p1; pC = p2; };
 if( p1in ){ pA = p1; pB = p0; pC = p2; };
 if( p2in ){ pA = p2; pB = p1; pC = p0; };

 px0 = PointOnPlane( pA /*vStart*/, pB /*vEnd*/,
 vPlaneNormal, vPointOnPlane );

 px1 = PointOnPlane( pA /*vStart*/, pC /*vEnd*/,
 vPlaneNormal, vPointOnPlane );

 String keyToSet = "";
 String keyToSet2 = "";
 //set keys and put intersection points in
 //HashMap
 if(map.containsKey(""+(double)px0.m_x * 10000+"
```

```
    "+(double)px0.m_y * 10000+"")){
 if(map.containsKey(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+" 1")){
 System.out.println("Corrupted Layer!!!");
 layers.get(layer).skipLayer = true;
 }else{
 if(map.containsKey(""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y *
    10000+""))map.put(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+" 1", new MapVector(new
    Vector3(px0.m_x, px0.m_y, px0.m_z), new
    Vector3(px1.m_x, px1.m_y,
    px1.m_z),""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y * 10000+" 1"));
 else map.put(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+" 1", new MapVector(new
    Vector3(px0.m_x, px0.m_y, px0.m_z), new
    Vector3(px1.m_x, px1.m_y,
    px1.m_z),""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y * 10000+""));
 list.add(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+" 1");
 keyToSet = ""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+" 1";
 }
}else{
 if(map.containsKey(""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y *
    10000+""))map.put(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+"", new MapVector(new
    Vector3(px0.m_x, px0.m_y, px0.m_z), new
    Vector3(px1.m_x, px1.m_y,
    px1.m_z),""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y * 10000+" 1"));
 else map.put(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+"", new MapVector(new
    Vector3(px0.m_x, px0.m_y, px0.m_z), new
    Vector3(px1.m_x, px1.m_y,
    px1.m_z),""+(double)px1.m_x*10000+"
    "+(double)px1.m_y * 10000+""));
 list.add(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y*10000+"");
 keyToSet = ""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+"";
}

if(map.containsKey(""+(double)px1.m_x * 10000+"
```

```
    "+(double)px1.m_y * 10000+"")){
 if(map.containsKey(""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y * 10000+" 1")){
  System.out.println("Corrupted Layer!!!");
  layers.get(layer).skipLayer=true;
 }else{
  map.put(""+(double)px1.m_x * 10000+"
     "+(double)px1.m_y * 10000+" 1", new MapVector(new
     Vector3(px1.m_x, px1.m_y, px1.m_z), new
     Vector3(px0.m_x, px0.m_y, px0.m_z),keyToSet));
  list.add(""+(double)px1.m_x * 10000+"
     "+(double)px1.m_y * 10000+" 1");
 }
}else{
 map.put(""+(double)px1.m_x * 10000+" "+(double)px1.m_y
    * 10000+"", new MapVector(new Vector3(px1.m_x,
    px1.m_y, px1.m_z), new Vector3(px0.m_x, px0.m_y,
    px0.m_z),keyToSet));
 list.add(""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y * 10000+"");
}
intersectionPointCnt++;
return 1;
}

if(iCount==2)
{

 Vector3 pA = new Vector3();
 Vector3 pB = new Vector3();
 Vector3 pC = new Vector3();


 if( !p0in ){ pA = p1; pB = p2; pC = p0; };
 if( !p1in ){ pA = p0; pB = p2; pC = p1; };
 if( !p2in ){ pA = p0; pB = p1; pC = p2; };


 px0 = PointOnPlane( pB /*vStart*/, pC /*vEnd*/,
     vPlaneNormal, vPointOnPlane );

 px1 = PointOnPlane( pA /*vStart*/, pC /*vEnd*/,
     vPlaneNormal, vPointOnPlane );

 String keyToSet = "";
 String keyToSet2 = "";
 //set keys and put intersection points in
```

```
//HashMap
if(map.containsKey(""+(double)px0.m_x * 10000+"
   "+(double)px0.m_y * 10000+"")){
 if(map.containsKey(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+" 1")){
  System.out.println("Corrupted Layer!!!");
  layers.get(layer).skipLayer = true;
 }else{
  if(map.containsKey(""+(double)px1.m_x * 10000+"
     "+(double)px1.m_y *
     10000+""))map.put(""+(double)px0.m_x * 10000+"
     "+(double)px0.m_y * 10000+" 1", new MapVector(new
     Vector3(px0.m_x, px0.m_y, px0.m_z), new
     Vector3(px1.m_x, px1.m_y,
     px1.m_z),""+(double)px1.m_x * 10000+"
     "+(double)px1.m_y * 10000+" 1"));
  else map.put(""+(double)px0.m_x * 10000+"
     "+(double)px0.m_y * 10000+" 1", new MapVector(new
     Vector3(px0.m_x, px0.m_y, px0.m_z), new
     Vector3(px1.m_x, px1.m_y,
     px1.m_z),""+(double)px1.m_x * 10000+"
     "+(double)px1.m_y * 10000+""));
  list.add(""+(double)px0.m_x * 10000+"
     "+(double)px0.m_y * 10000+" 1");
  keyToSet = ""+(double)px0.m_x * 10000+"
     "+(double)px0.m_y * 10000+" 1";
 }
}else{
 if(map.containsKey(""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y *
    10000+""))map.put(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+"", new MapVector(new
    Vector3(px0.m_x, px0.m_y, px0.m_z), new
    Vector3(px1.m_x, px1.m_y,
    px1.m_z),""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y * 10000+" 1"));
 else map.put(""+(double)px0.m_x * 10000+"
    "+(double)px0.m_y * 10000+"", new MapVector(new
    Vector3(px0.m_x, px0.m_y, px0.m_z), new
    Vector3(px1.m_x, px1.m_y,
    px1.m_z),""+(double)px1.m_x * 10000+"
    "+(double)px1.m_y * 10000+""));
  list.add(""+(double)px0.m_x * 10000+"
     "+(double)px0.m_y * 10000+"");
  keyToSet = ""+(double)px0.m_x * 10000+"
     "+(double)px0.m_y * 10000+"";
}
```

```
   if(map.containsKey(""+(double)px1.m_x * 10000+"
      "+(double)px1.m_y * 10000+"")){
    if(map.containsKey(""+(double)px1.m_x * 10000+"
       "+(double)px1.m_y * 10000+" 1")){
     System.out.println("Corrupted Layer!!!");
     layers.get(layer).skipLayer = true;
    }else{
     map.put(""+(double)px1.m_x * 10000+"
        "+(double)px1.m_y * 10000+" 1", new MapVector(new
        Vector3(px1.m_x, px1.m_y, px1.m_z), new
        Vector3(px0.m_x, px0.m_y, px0.m_z),keyToSet));
     list.add(""+(double)px1.m_x * 10000+"
        "+(double)px1.m_y * 10000+" 1");
    }
   }else{
    map.put(""+(double)px1.m_x * 10000+" "+(double)px1.m_y
       * 10000+"", new MapVector(new Vector3(px1.m_x,
       px1.m_y, px1.m_z), new Vector3(px0.m_x, px0.m_y,
       px0.m_z),keyToSet));
    list.add(""+(double)px1.m_x * 10000+"
       "+(double)px1.m_y * 10000+"");
   }
   intersectionPointCnt++;
   return 2;
  }
  return 3;
 }
}
```

Bibliography

[1] Mukesh K. Agarwala. 'Structural quality of parts processed by fused deposition'. In: *Rapid Prototyping Journal* 2 (1996), pp. 4–19.

[2] *Application of INVT Servo Motor to Engraving and Milling Machine (picture of milling machine)*. URL: http://www.invt.com/en/solution/cdetail.aspx?NC=119004001003002&ID=479.

[3] *CNC Milling: Introduction to cutting tools*. URL: http://wiki.imal.org/howto/cnc-milling-introduction-cutting-tools.

[4] *CNC Software for Non-Machinists*. URL: http://www.grzsoftware.com.

[5] CNCCookbook. *CAM Toolpath Strategies*. URL: http://www.cnccookbook.com/CCCAMToolpaths.htm.

[6] CNCCookbook. *CNC Dictionary*. URL: http://www.cnccookbook.com/CCDictionary.htm.

[7] CNCCookbook. *Coolant and Chip Clearing*. URL: http://www.cnccookbook.com/CCCNCMillFeedsSpeedsCoolant.htm.

[8] RS Components. *HE30TF Aluminium Rectangular Bar, 2in x 1in x 24in*. URL: http://no.rs-online.com/web/p/aluminium-rods-bars/4466763/.

[9] *DeskProto Website: press releases*. URL: http://www.deskproto.com/download/pressreleases.htm.

[10] *Disadvantages of Fused Deposition Modeling*. URL: http://www.buy3dprinter.org/3dprintingtechnologies/fused-deposition-modeling-fdm/.

[11] *Essential information about the DeskProto 3D CAM software*. URL: http://www.deskproto.com/products/essentials.htm.

[12] *Feed Rate Calculations*. URL: http://its.foxvalleytech.com/machshop3/speedcalc/feedratecalc.htm.

[13] Eclipse Foundation. *About the Eclipse Foundation*. URL: http://www.eclipse.org/org/.

[14] Matthew C. Frank. 'Implementing Rapid Prototyping Using CNC Machining (CNC-RP) Through a CAD/CAM Interface'. In: *Solid freeform fabrication symposium* 18 (2007), pp. 112–123.

[15] Lamson Global. *About Cutviewer*. URL: http://www.cutviewer.com/about/.

[16] Rodrigo MMH Gregori et al. 'Slicing Triangle Meshes: An Asymptotically Optimal Algorithm'. In: *Computational Science and Its Applications (ICCSA), 2014 14th International Conference on*. IEEE. 2014, pp. 252–255.

[17] Martin Held. 'A geometry-based investigation of the tool path generation for zigzag pocket machining'. English. In: *The Visual Computer* 7.5-6 (1991), pp. 296–308. ISSN: 0178-2789. DOI: 10.1007/BF01905694. URL: http://dx.doi.org/10.1007/BF01905694.

[18] Mats Høvin. *Lecture 1: Rapid Prototyping(RP) 1:6 -INF4500*. URL: http://heim.ifi.uio.no/matsh/inf4500/lec/0.php?s=6&l=1&d=0.

[19] Mats Høvin. *Lecture 11: G Code 11:29 -INF4500*. URL: http://heim.ifi.uio.no/matsh/inf4500/lec/0.php?s=29&l=11&d=0.

[20] Mats Høvin. *Lecture 11: Powder bed - inkjet head 3D printing (PBP) 9:13 -INF4500*. URL: http://heim.ifi.uio.no/matsh/inf4500/lec/0.php?s=13&l=9&d=0.

[21] Mats Høvin. *Lecture 12: Milling objectives and Milling strategy) 12:19,20 -INF4500*. URL: http://heim.ifi.uio.no/matsh/inf4500/lec/0.php?s=19&l=12&d=0.

[22] Mats Høvin. *Lecture 12: Work Holding: vice) 12:4 -INF4500*. URL: http://heim.ifi.uio.no/matsh/inf4500/lec/0.php?s=4&l=12&d=0.

[23] Mats Høvin. *Lecture 9: FDM tool path generation 9:8 -INF4500*. URL: http://heim.ifi.uio.no/matsh/inf4500/lec/0.php?s=8&l=9&d=0.

[24] Mats Høvin. *Robotics: Utvikling av rapid prototyping CNC fresemaskin (Thesis Description)*. URL: http://www.mn.uio.no/ifi/studier/masteroppgaver/robin/UtviklingAvRPcnc.html.

[25] *How Fused Deposition Modeling (FDM) Works*. URL: https://thre3d.com/how-it-works/material-extrusion/fused-deposition-modeling-fdm.

[26] HSMWorks. *Integrated CAM for SolidWorks*. URL: http://www.hsmworks.com/overview/.

[27] *Image of flat mill*. URL: http://3.imimg.com/data3/IA/CR/MY-1164707/end-mills-for-woodworking-250x250.jpg.

[28] T. Insperger et al. 'Stability of up-milling and down-milling, part 1: alternative analytical methods'. In: *International Journal of Machine Tools and Manufacture* 43.1 (2003), pp. 25–34. ISSN: 0890-6955. DOI: http://dx.doi.org/10.1016/S0890-6955(02)00159-1. URL: http://www.sciencedirect.com/science/article/pii/S0890695502001591.

[29] Ali Kamrani. *Rapid Prototyping: Theory and Practice*. Springer US, 2006, pp. 174–175.

[30] Ben Kenwright. *Clipping - Slicing using a plane*. URL: http://www.xbdev.net/java/tutorials_3d/clipping/index.php.

[31] Bo H Kim and Byoung K Choi. 'Machining efficiency comparison direction-parallel tool path with contour-parallel tool path'. In: *Computer-Aided Design* 34.2 (2002), pp. 89–95.

[32] Hyun-Chul Kim. 'Tool path generation for contour parallel milling with incomplete mesh model'. English. In: *The International Journal of Advanced Manufacturing Technology* 48.5-8 (2010), pp. 443–454. ISSN: 0268-3768. DOI: 10.1007/s00170-008-1733-9. URL: http://dx.doi.org/10.1007/s00170-008-1733-9.

[33] Ali Lasemi, Deyi Xue and Peihua Gu. 'Recent development in {CNC} machining of freeform surfaces: A state-of-the-art review'. In: *Computer-Aided Design* 42.7 (2010), pp. 641–654. ISSN: 0010-4485. DOI: http://dx.doi.org/10.1016/j.cad.2010.04.002. URL: http://www.sciencedirect.com/science/article/pii/S0010448510000631.

[34] Lex Lennings. 'An introduction to CAM Software'. In: *Desktop CNC comparison table website (no loger exitsing)* (May, 2001).

[35] Lex Lennings. 'RP with CNC'. In: *Prototyping Technology Intenational 99* (1999), was never published.

[36] Carbide 3D LLC. *The Nomad CNC Mill*. URL: https://www.kickstarter.com/projects/178590870/the-nomad-cnc-mill.

[37] *Milling Step-over Distance Calculator*. URL: http://www.custompartnet.com/calculator/step-over-distance.

[38] Oracle. *Design Goals of the Java Programming Language*. URL: http://www.oracle.com/technetwork/java/intro-141325.html.

[39] Oracle. *What is Java*. URL: https://www.java.com/en/download/faq/whatis_java.xml.

[40] Shi Hyoung Ryu, Hae Sung Lee and Chong Nam Chu. 'The form error prediction in side wall machining considering tool deflection'. In: *International Journal of Machine Tools and Manufacture* 43.14 (2003), pp. 1405–1411.

[41] Shapeoko. *Climb vs. Conventional Milling*. URL: http://www.shapeoko.com/wiki/index.php/Climb_vs._Conventional_Milling.

[42] Lars Skaret. 'A Stewart Platform Based Replicating Rapid Prototyping System with Biologicallt Inspired Path-Optimization'. MA thesis. Department of Informatics, May 2, 2011.

[43] Vivid Solutions. *JTS Topology Suite Technical Specifications*. URL: http://www.vividsolutions.com/jts/JTSHome.htm.

[44] Yong-Ak Song et al. '3D welding and milling: Part I–a direct approach for freeform fabrication of metallic prototypes'. In: *International Journal of Machine Tools and Manufacture* 45.9 (2005), pp. 1057–1062.

[45] Torjus Spilling. 'Self-Improving CNC Milling Machine'. MA thesis. Department of Physics, 2014.

[46] Kamesh Tata. 'Efficient slicing for layered manufactering'. In: *Rapid Prototyping Journal* 4 (1998), pp. 151–167.

[47] *Update on Carbide LLC and their Nomad CNC Mill*. URL: http://blog.cnccookbook.com/2014/08/29/update-carbide-llc-nomad-cnc-mill/.

[48] github wiki. *About GRBL*. URL: https://github.com/grbl/grbl/wiki.

[49] Will Winder. *Universal G-code Sender README*. URL: https://github.com/winder/Universal-G-Code-Sender/blob/master/README.md.

[50] Will Winder. *Universal G-Code Sender v1.0.5 released!* URL: http://www.shapeoko.com/forum/viewtopic.php?f=6&t=1053.

[51] ZH Xiong, CG Zhuang and H Ding. 'Curvilinear tool path generation for pocket machining'. In: *Proceedings of the Institution of Mechanical Engineers, Part B: Journal of Engineering Manufacture* 225.4 (2011), pp. 483–495.